To create a filler grammar, use the GSL keyword *filler*. For example, the following creates a filler grammar defining the phrases “I want to” and “I wish to:”

```
IWantTo:filler      [(i want to) (i wish to)]
```

You could use this grammar as follows:

```
.Main (?IWantTo travel by car)
```

You have access to filler/non-filler scoring differences at runtime, so that you can compare recognition performance. See the online documentation on the functions `RecResultOverallConfidenceWithoutFiller` and `RecResultOverallConfidence`.

Tuning recognition performance

The accuracy and speed of the Nuance recognizer are determined by several factors:

- *The input speech: noise level, distortions, speech clarity, and so on*

  The cleaner the speech sample, the faster and more accurate the recognition.

- *The dialog design*

  Recognition is hurt when the speaker is unsure of what to say. A good dialog design dispels any doubts on the part of the speaker as to what can or should be said in a dialog.

- *The complexity of the grammar*

  A large or complex grammar usually results in a slower recognition system. Recognition may be less accurate because of the larger number of possible word sequences, or may be more accurate if the grammar better reflects actual input speech.

- *The confusability of the grammar*

  Grammars that depend on the ability to distinguish between words that sound similar, such as “John Smith” and “Jon Smits,” typically result in higher error rates.

- *The complexity of the acoustic models*

  More complex models are more accurate but can result in slower recognition.

- *The use of the crossword option*

  Use of the option *-do_crossword* to compile a grammar, will improve accuracy. However, it may result in slower recognition performance. Most gains are

accomplished when used for small grammars with short words and many word boundaries, such as grammars for digits, alpha-digits and currency.

Nuance recommends that you turn the crossword option on for grammars where recognition performance is not an issue. This recommendation is true regardless of the language, but specially true for those languages with many short words, like Cantonese. The best way to determine if this is an appropriate option for your grammar is to try it out and compare results.

- *The amount of search performed*

  The recognition system can be configured to perform a wider or narrower search. Less searching speeds up the system, but might increase the error rate if good theories are missed.

Sound grammar and application design are essential to good recognition performance. The performance of the Nuance recognizer can be further optimized by varying system configurations and by using more specialized grammar specification techniques. These include:

- Experimenting with different acoustic models
- Tuning parameter settings
- Creating more accurate pronunciations
- Using grammar probabilities

When developing and refining an application, Nuance recommends that you collect recordings of application usage and use them to measure the speed and accuracy of the recognizer offline under various configurations using the *batchrec* program. This tool is briefly described in "Command-line tools for grammar testing" on page 68.

Summary of compiler options

The table below shows some of the optional arguments to *nuance-compile*, along with a brief description. For a complete listing of options, see the online documentation.

Table 4: *nuance-compile* options

Option	Notes
-merge_dictionary *filename*	Merges standard dictionary entries with custom dictionary entries.

Table 4: *nuance-compile* options

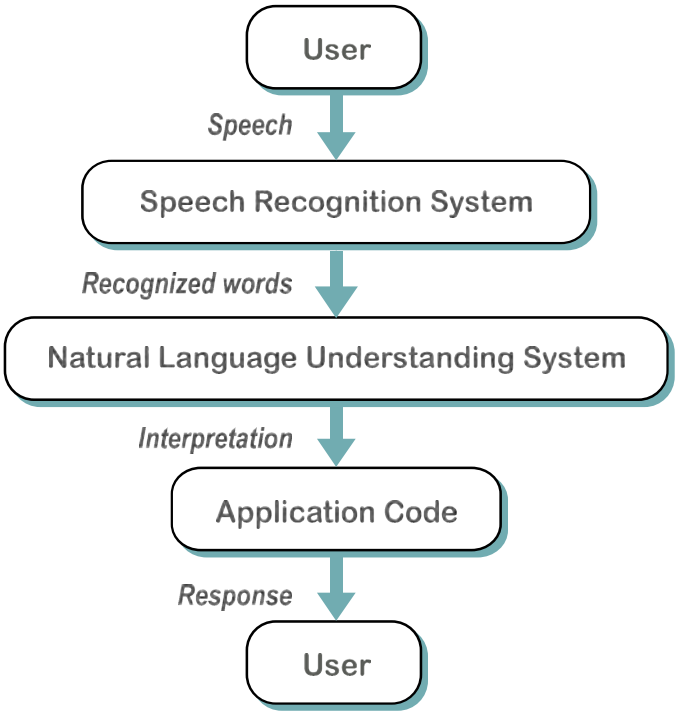
Option	Notes
-override_dictionary *filename*	Replaces master dictionary entries with custom dictionary entries.
-auto_pron	Enables the generation of pronunciations for missing words. The generated pronunciations are written, by default, to file *package.autopron.*
-nooverwrite	Prevents deletion of previously compiled files.
-write_auto_pron_output *filename*	Writes the generated pronunciations to a specified file. It is meaningful only when used with the option -auto_pron.
-optimize_graph	Enables graph optimization pass (default). Leads to faster recognition but takes longer to compile.
-dont_optimize_graph	Disables graph optimizations. Speeds up compilation, but may slow down runtime performance.
-dont_use_grammar_probs	Tells the compiler to omit grammar probabilities from the compiled package. Useful to compare performance of package with and without probability specifications.
-dont_flatten	Expands each grammar only once during compilation. Produces smaller binary; limits the generation of compound-word pronunciations; must be used with recursive grammars.
-o *name*	Overwrites default package output name.
-options	Displays all compiler options available. Use with no other option.

Chapter 4

Natural language understanding

4

The Nuance System lets you define natural language understanding in your recognition packages. A natural language understanding system takes a sentence (typically a recognized utterance) as input and returns an *interpretation*—a representation of the meaning of the sentence. The application code can then take action based on the user's request:



The natural language system simplifies the work your application needs to do to correctly respond to user utterances. Often, there are many ways a user can express the same meaning. For example, the following phrases:

"Withdraw fifteen hundred dollars from savings"
"Take fifteen hundred out of savings"
"Give me one thousand five hundred dollars from my savings account"

map to the same semantic units:

action = withdraw
amount = 1500.00
account = savings

The Nuance natural language understanding engine returns a simple, structured interpretation, based on a predefined set of "slots" with associated "values." Applications can then access this interpretation directly without needing to parse the recognized text. You define the natural language commands for an application right in the grammar file, making it easy to update your definitions and recompile the recognition package.

Defining natural language interpretations

To define natural language understanding for a recognition package, you:

- Define a fixed set of slots that correspond to the types of information supplied in utterances in the application's domain
- Determine how each phrase in the grammar causes slots to be filled with specific values

In an automated banking application, for example, slots might include *command-type*, *amount*, *source-account*, and *destination-account*. Each slot has an allowed set of values—the *source-account* slot could be filled with the strings "checking," "savings," "money market," while the *amount* slot is filled with a numeric value, and so on.

The utterance "Transfer five hundred dollars from savings to checking" would generate an interpretation structure with the following values:

command-type = transfer
source-account = savings
destination-account = checking
amount = 500

Natural language commands

You add natural language understanding capability to a grammar by defining the set of supported natural language *slots*, and then including special natural language *commands* in the grammar file that map values to those slots. A natural

language command is specified within curly braces ({ and }) , and attaches to the grammar construct immediately preceding it. For example, in the next grammar, the command attaches to the word "from":

```
Phrase ( from {...} checking )
```

while in the next one, the command attaches to the entire OR construction, [checking savings]:

```
Phrase ( from [ checking savings ] {...} )
```

There are two kinds of natural language commands:

- A *slot-filling* command that indicates a slot is to be filled with a particular value
- A *return* command that lets you associate a particular value with a grammar without actually filling a slot

Slot-filling commands

A slot-filling command has the form:

<Slot Value>

The `Phrase1` grammar below uses slot-filling commands to specify the string with which the *source-account* slot is filled:

```
Phrase1 (from [
        (?my checking ?account) {<source-account checking>}
        (?my savings ?account)  {<source-account savings>}
        ] )
```

Note: Notice that although "checking" can be expressed in four possible ways ("*checking*", "*checking account*", "*my checking account*", and "*my checking*"), the *source-account* slot is always filled with the value "checking."

In addition to string values, a slot can also be filled with an integer value, as in the next example:

```
Phrase2 ( transfer
        [ fifty   {<transfer-amt 50>}
          sixty   {<transfer-amt 60>}
        ]
        dollars )
```

If the natural language system can treat a value as an integer, it does so. You can force a value to be treated as a string by enclosing it in double quotes. You might want to do this in digit grammars, where treating the value as an integer causes leading zeroes to be omitted. So instead of creating a digit grammar with constructs like:

```
( zero zero zero )  {<digits 000>}
( zero zero one )    {<digits 001>}
```

you should create a grammar like the following:

```
Digits [ ( zero zero zero ) {<digits "000">}
         ( zero zero one )  {<digits "001">}
         ( zero zero two )  {<digits "002">}
         . . .
]
```

You can also fill multiple slots within a single natural language command. The `Phrase3` grammar below fills two slots:

```
Phrase3 ( from savings to checking )
    {<source-account savings> <destination-account checking>}
```

Return commands and variables

Return commands let you associate values with a grammar, without filling any slots. The commands in the `Account` grammar below cause one of the string values "checking" or "savings" to be returned:

```
Account ( ?my
        [checking {return(checking)}
        savings {return(savings)}]
        ?account )
```

Return values do not appear in the interpretation structures produced by the natural language understanding system—you must use the return value to fill a slot. You do this by using *variables*. In your grammar specification, you assign the return value of a grammar to a variable, and then reference that variable to fill a slot in a slot-filling command.

Assigning variables

To assign the return value of a grammar to a variable, follow the grammar name with a colon and the variable name, as in:

```
( from Account:acct )
```

where *Account* is the grammar described above, and *acct* is the variable. In this case, the variable *acct* is assigned one of the values "checking" or "savings." You can then reference the value of a variable in a slot-filling command using the "$" character:

```
Phrase4 ( from Account:acct ) {<source-account $acct>}
```

The variable *acct* is filled with the return value of the *Account* grammar, and that value is then referenced to fill the slot *source-account*.

Caution: If you omit the '$' before the variable name, the slot is filled with the string "acct" rather than the value of the variable.

Variables must be set before they can be referenced. In the following grammar the *source-account* slot is *not* filled:

```
Phrase5 ( from {<source-account $source>} Account:source )
```

Return values are most useful when a certain type of value is used to fill multiple slots. In the `Phrase6` grammar below the variable `acct` is used to specify how both the *source-account* and *destination-account* slots get filled:

```
Phrase6 [   ( from Account:acct) {<source-account $acct>}
            ( to Account:acct ) {<destination-account $acct>}
        ]
```

Variable names can only contain alphanumeric characters and the following special characters: dash (-), underscore (_), single quote (’), and “at” sign (@).

When returning a string value, enclosing the character sequence between double quotes is optional. Thus both expressions below are valid and equivalent:

```
{ return(cat) }
{ return(“cat”) }
```

However, the use of double quotes is encouraged to clearly differentiate the expressions:

```
{ return(1) }
{ return(“1”) }
```

where the first one returns an integer and the second a string.

The *string* variable

The variable mechanism also provides a way to use the actual string of spoken words to fill a slot instead of explicitly specifying the string value to return. You can do this using the special variable *string*. When you use the *string* variable, the slot is filled with the portion of the input utterance that matched that grammar. Consider the following grammar:

```
Day1 [ sunday monday tuesday wednesday thursday friday saturday
     ] {return($string)}
```

The special variable *string* allows you to write cleaner specifications than specifying each return value, as in:

```
Day2 [  sunday {return(sunday)}
        monday {return(monday)}
        . . . ]
```

How slots are filled

At runtime, the natural language system generates interpretations by matching the input utterance with a phrase (defined by a grammar) and executing any natural language commands attached to that grammar (subsequently referred to as the *matching* grammar).

Whenever a matching grammar has a construct with a natural language command attached to it, that command is executed and slot/value pairs are added to the interpretation. Commands attached to a grammar construct that is not matched by the utterance are not executed and, therefore, no slot/value pair is added to the interpretation.

For example, if the utterance “from savings” is processed against the following grammar:

```
Phrase [ ( from Account:acct ) {<source-account $acct>}
         ( to Account:acct ) {<destination-account $acct>}
       ]
```

the *source-account* slot is filled, but the *destination-account* slot is not.

When defining slot-filling commands, keep in mind the following points:

- *A variable must be assigned before it is referenced*

  If a slot-filling command is executed but contains a variable that is not assigned, the slot is not filled. So in the next grammar example, the *source-account* slot is not filled when the input utterance is "to checking":

  ```
  Banking [(from Account:source)
           (to Account:dest)
          ] { <source-account $source>
              <destination-account $dest>}
  ```

- *Do not fill a slot with more than one value*

  No interpretation is produced if a slot is filled with more than one value in the matching grammar. The runtime system generates a warning when it finds a construct in a matching grammar that violates this principle.

- *Commands have precedence over the unary operators*

  This rule resolves how constructs preceded by a unary operator (?, * or +) and followed by a command are parsed.

  Consider the following grammar:

  ```
  ImperativeGo ( ?please {<polite yes>} go )
  ```

  If "please" is *not* present in the utterance, then the slot `polite` is not filled. This is because the command is attached to the construction `please` and not the construction `?please`. If the command were attached to the construction `?please`, the slot would be filled whether the input is "go" or "please go," as in:

  ```
  ImperativeGo ( (?please) {<polite yes>} go )
  ```

The slot definitions file

When you create a project within Nuance Grammar Builder, a slot definitions file is automatically included in the project. If the project's name is *myProject*, then the slot definitions file is named *myProject_master.slot_definitions*.

You must add all slots used in any grammar included in *myProject* to this file. This file is shown in the tree project. Double-clicking its icon brings up the contents of the file in a editor window. This file cannot be deleted from a project.

The slot definitions file is a plain text file that lists all the slot names used in your grammars, one per line. A slot name used in a grammar that is not listed in the slot definitions file will cause a compilation error.

For the `Phrase` grammar in the preceding section, the slot definitions file would contain the following list:

```
source-account
destination-account
```

As in a grammar file, any line in a slot definitions file that begins with a semicolon is considered a comment and therefore ignored by the grammar compiler.

Note: If you are using the command-line utilities, create the file manually. You must name it *name.slot_definitions*, where *name* is the name of the recognition package that you are defining. It must reside in the same directory as your grammar files.

Compiling a grammar with natural language support

The grammar compilation program *nuance-compile*, described in Chapter 3, automatically compiles natural language capabilities into the recognition package when the directory contains a *slot_definitions* file.

Nuance also provides the tool *nl-compile* that compiles only enough of the recognition package to support interpretation rather than providing full support for speech recognition. You can use this program to quickly compile grammars for natural language testing.

To use *nl-compile*, just specify the package name, for example:

```
> nl-compile %NUANCE%\sample-packages\banking1
```

Ambiguous grammars

A grammar is ambiguous if a sequence of words can produce multiple interpretations. For example, the following grammar:

```
.Command (call Name:nm) {<command call> <name $nm>}
Name [ [john (john smith)]      {return(john_smith)}
       [mary (mary jones)]      {return(mary_jones)}
       [john (john brown)]      {return(john_brown)}
        . . .
     ]
```

is ambiguous because the word sequence “call john” produces two interpretations:

```
{<command call> <name john_smith>}
{<command call> <name john_brown>}
```

When ambiguous sentences occur, the natural language system returns multiple interpretations, sorted by probability (if specified). See “Grammar probabilities” on page 25 for information on specifying probabilities in a grammar.

In some situations you may not be able to avoid ambiguity in your grammars—your application will have to resolve the ambiguity by asking the caller which interpretation was intended, by using:

- Program logic
- A subdialog

In the NGB environment, once you have compiled your project, you can determine whether your grammar produces any ambiguous phrases as follows:

1. Click the Generate icon on the Grammar toolbar or, alternatively, select the Generate Sentences from the Tools menu. This brings up the Generate Options dialog
2. Select the top-level grammar to test, and check the Ambiguous Only box
3. Click OK

NGB produces a test set window with the list of all ambiguous phrases in your grammar. This test set is empty if no ambiguity is found in your grammar.

To test your grammar for ambiguity from the command line, use the program *generate*, described in “Command-line tools for grammar testing” on page 68.

Advanced features

You can create many speech recognition applications that incorporate natural language understanding using only the features described in “Defining natural language interpretations” on page 46. However, some applications require more advanced features, such as:

- Filling a slot with a function of multiple values
- Filling a slot with a complex data type

Functions

In the grammars described so far, all slot and return values have been constants—either strings or integers. The natural language specification system also lets you fill a slot or specify a return value with a *non-constant* function. GSL

supports a standard set of functions for manipulating integers and strings, and you can define your own functions.

Standard functions

GSL supports standard functions for filling a slot or specifying a return value, based on simple integer arithmetic or string concatenation. The following grammar demonstrates how you can specify a return value that is the sum of two other return values:

```
NonZeroDigit [ one      {return(1)}
               two      {return(2)}
               three    {return(3)}
               four     {return(4)}
               five     {return(5)}
               six      {return(6)}
               seven    {return(7)}
               eight    {return(8)}
               nine     {return(9)}
              ]
TwentyToNinety [ twenty     {return(20)}
                 thirty     {return(30)}
                 forty      {return(40)}
                 fifty      {return(50)}
                 sixty      {return(60)}
                 seventy    {return(70)}
                 eighty     {return(80)}
                 ninety     {return(90)}
                ]
TwoDigit [ (    TwentyToNinety:num1 NonZeroDigit:num2 )
                        {return(add($num1 $num2))}
                TwentyToNinety:num1
                        {return($num1)}
        ]
```

This grammar creates return values for numbers between 20 and 99 by adding the values returned by two subgrammars, *NonZeroDigit* and *TwentyToNinety*. For example, for the utterance “fifty nine,” the *TwentyToNinety* grammar returns 50, and the *NonZeroDigit* grammar returns 9. These are assigned to variables, which the grammar *TwoDigit* then adds to return the value 59.

The function mechanism can also handle variables that have not been set, as in the following alternative specification of the *TwoDigit* grammar:

```
TwoDigit2 ( TwentyToNinety:num1 ?NonZeroDigit:num2 )
                    {return(add($num1 $num2))}
```

When the grammar `TwoDigits2` processes an utterance such as “thirty,” the `add` function substitutes 0 for the variable *num2* and correctly returns the value 30.

The following table lists the available standard functions for integer and string manipulation, and describes how each function treats a non-set variable:

Table 5: Standard integer and string functions

Function	Description	Unset variable
add	Returns the sum of two integers.	0
sub	Returns the result of subtracting the second integer from the first.	0
mul	Returns the product of two integers.	1
div	Returns the truncated integer result of dividing the first integer by the second (e.g., div(9 5) evaluates to 1).	0 if first argument; 1 if second argument
neg	Returns the negative counterpart of a positive integer argument, or the positive counterpart of a negative integer argument.	0
strcat	Returns the concatenation of two strings. This is a binary operator, but it accepts nested calls as in strcat($a1 strcat($a2 $a3)), which effectively concatenates three strings.	"" (empty sting)

You can create more complex specifications by composing standard functions. The grammar `ThreeDigit` below builds on the previous `TwoDigit` grammar to cover three-digit numbers spoken in the form "two twenty three":

```
ThreeDigit ( NonZeroDigit:num1 TwoDigit:num2 )
                    {return(add(mul($num1 100) $num2))}
```

For examples of the use of functions in NGB see, for instance, the grammar *time.grammar* under the *Time* folder in the Grammar Library.

Note: See *%NUANCE%\data\lang\English.America\grammars\number.grammar* for an example of a grammar file that makes extensive use of the standard functions.

User-defined functions

The grammar syntax also supports the creation of your own functions for use in generating slot values. You define your functions in a file called *name.functions* (where *name* is the name of the package you are defining) that you include in the directory with your other grammar files, and it is processed during compilation by *nuance-compile* or *nl-compile*.

To define a function, specify the possible arguments it can take and the values it yields on those arguments. The function below, for instance, maps a day of the week to the following day of the week:

```
next_day ( <sunday      monday>
           <monday      tuesday>
```

```
    <tuesday     wednesday>
    <wednesday   thursday>
    <thursday    friday>
    <friday      saturday>
    <saturday    sunday>
  )
```

The `next_day` function takes a single string argument—you can also define functions that take more arguments. The following function sample maps a month and year into the number of days in that month in that year:

```
days_in_month ( <january 1999 31>
                 <january 2000 31>
                 <february 1999 28>
                 <february 2000 29>
                etc.
               )
```

Each triplet delimited by angle brackets in this function specifies one possible group of arguments to the function and the result of the function when applied to those arguments. The function result is always the last value in the set, regardless of the number of arguments.

You use user-defined functions in the same way as standard functions. You might use the `next_day` function in a grammar as follows:

```
DaySpec ( the day after DayOfWeek:dow ) {return(next_day($dow))}
```

Complex values

Strings and integers are the two types of simple values supported by the natural language system. The system also supports two types of *complex* values:

- Structures, which contain a set of slot/value pairs
- Lists, which let you fill a slot with a set of values

A value used in a structure or list may be either of simple or complex type, and the values in a list or structure don't need to be of same kind; thus a list could consist of an integer, a string, and another list.

Structures

A structure lets you create values with multiple name/value pairs. Suppose you want to create a date grammar that simultaneously returns values representing the month, day, and year. This is most naturally represented as a set of slot/value pairs whose elements are:

```
<month may>, <day 15>, <year 1995>
```

The following simple date grammar illustrates how to return a structure:

```
Month [ january february march april
        may june july august september
        october november december ] {return($string)}
```

```
Day [ first     {return(1)}
      second    {return(2)}
    etc.
]
Year [ ( nineteen ninety five ) {return(1995)}
    etc.
]
Date ( Month:m Day:d ?Year:y )
                {return([<month $m> <day $d> <year $y>])}
```

Each slot within a structure is referred to as a *feature*, and has a value that is a string, integer, or another structure.

Referencing feature values

You can create specifications that reference values of features within a structure through variable expressions.

Assume that a variable, *d*, is set to a structure that has *month*, *day*, and *year* features, and that you want to include the value of the *month* feature of *d* in a slot called *depart-month*. The following grammar illustrates how to do so:

```
Phrase (departing on Date:d) {<depart-month $d.month>}
```

The expression `$d.month` evaluates to the value of the *month* feature of the structure to which the variable *d* refers. (If *d* has no *month* feature, then the expression `$d.month` has no value, and the slot *depart-month* does not get filled.)

The grammar *DateConstraint* below illustrates a realistic use of the structure returned by the previous `Date` grammar:

```
DateConstraint [
        (departing on Date:dep-date)
            {<depart-month $dep-date.month>
             <depart-day $dep-date.day>
             <depart-year $dep-date.year>}
        (returning on Date:ret-date)
            {<return-month $ret-date.month>
             <return-day $ret-date.day>
             <return-year $ret-date.year>}
]
```

Although there is a single *Date* grammar, the structure returned is used to fill the departing and returning slots. This kind of specification would not be possible without using structures.

You can also fill slots with entire structures, rather than components of those structures. For example:

```
DateConstraint [
        (departing on Date:date) {<depart-date $date>}
        (returning on Date:date) {<return-date $date>}
]
```

The application handling the interpretation can access the value of each feature within a structure using the natural language API functions. Consult the online documentation for information on natural language APIs.

Nested structures

You can create specifications that include nested structures. For example, the structure:

```
[<time 1100>
 <date [<month may> <day 16> <year 1995>]>
]
```

includes a `date` structure as a feature value of a larger structure. This grammar illustrates how you can access the *month* feature of the *date* feature of that structure:

```
Phrase ( departing on TimeDate:time-date )
                {<depart-month $time-date.date.month>}
```

The expression `$time-date.date.month` evaluates to the value of the *month* feature of the *date* feature of the structure referred to by *time-date*. If the utterance is "on May 16th, 1999 at 6 AM" this feature has the value "may." If *time-date* has no *date* feature, or if its *date* feature has no *month* feature, then the slot *depart-month* is not filled.

Several grammars in the Grammar Library make extensive use of structures. See the file *date.grammar*, under the *Date* folder, or the file *time.grammar* under the *Time* folder.

Note: If you aren't using NGB, see the grammar file *date.grammar* in *%NUANCE%\data\lang\English.America* for the complete contents of a grammar using nested structures.

Lists

Using lists, you can fill a slot or return value with a sequence of values. Here is a simple example of a grammar that uses a list:

```
DigitString ( one two three ) {<digit_slot (1 2 3)>}
```

When this grammar matches the input string "*one two three,*" it sets the slot *digit_slot* to a list containing the integers 1, 2, and 3. In general, you can fill a value with a list of any sequence of values by enclosing the sequence between parentheses.

You can construct lists explicitly as well as using more complex mechanisms such as functions and commands, described in the following sections.

List functions

The natural language system includes a number of built-in functions you can use to construct list values. The following example illustrates a grammar that matches any three-digit string:

```
Digit [ one {return(1)}
        two {return(2)}
```

```
    etc.
]
TwoDigit ( Digit:d1 Digit:d2 ) {return(($d1 $d2))}
ThreeDigit ( Digit:d TwoDigit:list )
                    {return(insert-begin($list $d))}
```

The `insert-begin` function creates a new list by adding an item to the beginning of an existing list. If the utterance is now "one four five," the *ThreeDigit* grammar returns a value by inserting "1" before the two-digit list "4 5" returned by the *TwoDigit* grammar.

The following table lists all the functions provided for working with lists:

Table 6: Built-in list functions

Function	Description
insert-begin	Returns a list with an item inserted at the beginning
insert-end	Returns a list with an item inserted at the end
concat	Returns the concatenation of two lists
first	Returns the first item in a list
last	Returns the last item in a list
rest	Returns a list with the first item removed

As with the integer and string functions, you can apply the list functions *insert-begin, insert-end,* and *concat* to arguments that are unset variables. In each case, the unset variable is treated as if it were an empty list.

Caution: It is an error to apply *first, last,* or *rest* to an unset variable, a non-list argument, or an empty list.

List commands

Several of the list functions can also be used as commands when constructing list values. Commands differ from functions in that:

- Commands appear within curly braces and perform an action. For example, the following command fills a slot:
  `{<month may>}`
- Functions merely specify a value and must be used within a command to have any effect. For instance, the following function specifies a list value:
  `concat($list1 $list2)`

  To use the value returned by this function, it must be used within a command:
  `{return (concat($list1 $list2))}`

List commands differ from return and slot-filling commands in that they actually change the value of a variable. The grammar below uses a list command to create a list value that matches a digit string of arbitrary length:

```
DigitString +( Digit:d {insert-end(list $d)} )
    {return($list)}
```

As the grammar continues to match additional digits, it updates the value of the variable *d* until the full list is created and returned, representing the spoken string of digits in the appropriate order.

The *insert-end* command differs from the *insert-end* function in that the command *changes* the value of the list passed to it by inserting the value of the second argument at the end of the list. Like the *insert-end* function, the command treats an unset variable as an empty list.

This table lists the supported list commands:

Table 7: List commands

Command	Description
insert-begin	Modifies a list variable by inserting an item at the beginning
insert-end	Modifies a list variable by inserting an item at the end
concat	Modifies a list variable by concatenating it with the contents of another list

These commands modify the value of the first argument. So the command

```
insert-begin(list1 2)
```

modifies the value of the variable *list1*, adding the value 2 to the beginning of the list:

Note: Note that the first argument is a variable, *not* a reference to a variable. So *insert-end(list 2)* executes the command, whereas *insert-end($list 2)* is an instance of the function.

Chapter 5

Testing grammars

5

A grammar definition is “code” and, therefore, prone to have bugs just like any other piece of software or data. This chapter presents a methodology for testing a grammar that will help you find and fix bugs in your grammars.

The natural tool to conduct the tests described in this chapter is Nuance Grammar Builder. You can also use command-line programs for your testing.

The first section presents an overview of what it means to test a grammar. The material in the second section is entirely dedicated to the description of test procedures within the NGB’s environment. The third section lists the command-line programs that Nuance provides to conduct basic testing. Typically, these programs are used in scripts that you create and run to test your grammars.

Grammar testing overview

A good plan for testing a grammar should include the following kind of tests:

Coverage test

The goal of a coverage test is to verify that your grammar is able to parse a prescribed set of phrases. Usually you maintain a set of phrases—and, possibly, interpretations—that you require your grammar to parse after any change is introduced or any compilation parameter is modified.

Interpretation test

This test verifies that slots are filled accurately and that all grammars return the correct values—that is, that your grammar delivers the expected natural language interpretation for a prescribed collection of phrases.

Over-generation test

This test is intended to expose phrases that should *not* be parsed by your grammar. It helps you verify that your grammar will not accept unwanted sentences.

Ambiguity test

This test exposes phrases parsed by your grammar that have multiple interpretations.

Pronunciation test

This test is used to detect words with unknown pronunciations. It also exposes misspellings in your grammars.

Testing a grammar with Nuance Grammar Builder

A special NGB project, called *tutorial*, is shipped with the tool and is used throughout this section to exemplify the concepts being discussed. The section demonstrates how you run tests to find and fix grammar errors that are intentionally included in the project.

Creating an initial test set

Before you write a new grammar, it is important to create an initial test set containing a number of the phrases you expect your grammar to generate. Here are a few suggestions to help you design your initial test set.

- *Have the prompt wording in mind.*

  Different prompts elicit different responses from the users. For example, if the prompt asks “From what city are you traveling?”, the test set should include the following phrases:

  - i am traveling from san jose
  - we are traveling from san jose
  - um san jose
  - from san jose
  - leaving from san jose
  - san jose california

  If, instead, the prompt is phrased “What is your departure city?”, then the test set should include the following phrases:

- san jose
- my departure city is san jose
- it is san jose
- it's san jose
- uh it is san jose
- the departure city is san jose
- san jose california

- *Do not generate your initial test set directly from the grammar.*

  Your grammar may already contain errors and you also won't be able to detect whether you have achieved the desired level of coverage. Rather, you should generate the initial test set by hand.

  After you are sure that your grammar is correct and parses a set of desired phrases, you can fill out the test set with phrases generated by the grammar. As your grammar evolves, it is also important that you keep the test set up to date with the grammar.

- *When your grammar contains a long list of items, it is best to include the entire list in your initial test set, if at all possible.*

  If you have more than a few thousand items, though, this may not be practical. Keep the test set up to date with changes in the list. That way, you will notice quickly if you have introduced any errors related to the items on the list.

Step-by-step grammar testing instructions

This section outlines a complete guide to testing a grammar with Nuance Grammar Builder that follows the strategy described in "Grammar testing overview" on page 61. The material used is all contained within the tool's environment. As you follow the instructions, you will find and fix typical errors that have been planted in a grammar file.

Open the project

Open the tutorial project in NGB, by selecting the file called *tutorial.nuance* located in the directory *%NUANCE%\Grammar Builder\Tutorial.*

If you do not have NGB running yet, you can start NGB *and* open this project at once by double-clicking the file *tutorial.nuance* in the Windows Explorer or file browser.

Compile the grammar

1 From the Tools menu, select the Compile command.

The Compile Grammar command is also invoked by clicking the first icon on the left in the Grammar toolbar or by pressing the F7 function key.

2 Select *English.America.1* from the pull down list in the Master Package box.

3 Leave the Auto Pron box unchecked. This prevents the compiler from performing automatic pronunciation generation.

4 Click OK to start the compilation.

The compiler lists the files it is processing and the results of the compilation in the output window. A slot definition is missing, so you should see the following error message in the output window:

```
Undefined slot name in grammar: "origin"!
```

Define the missing slot

1 Open the project file *tutorial_master.slot_definitions* by double-clicking its icon in the project tree view.

2 Follow the directions in the file to fix the bug—uncomment the line that contains the word `origin`.

3 Save the slot definitions file.

Recompile and find the second error

1 Compile the grammar once again, making sure the Auto Pron box is *not* checked.

This time, you should see the following warning message (among others):

```
Warning: There are 2 words missing from the dictionary.
```

Actually, the problem is not that there are words missing from the dictionary, but that there are typos in the grammar file.

2 Double-click on the error message to view the *tutorial_master.missing* file. Notice that it contains two words "columnbus" (a misspelling of the word "columbus") and "warshington" (a misspelling of "washington").

3 Open the *departure_city.grammar* file.

4 Locate the word "columnbus" (try searching for it) in that grammar file.

5 Change the words "columnbus" to "columbus" and "warshington" to "washington", and save the changes.

6 Recompile with the shown options—identical to those used for the previous compilation.

This time the grammar should compile with no errors.

Open the test set

For the next several procedures you need a test set. A small test set for the .DEPARTURE_CITY grammar has been provided.

1. Locate the file *tutorial.grammar_test_set* in the project window under the folder labeled Test Set Files and double-click it to display its contents.
2. Inspect the phrases under the Test Set column—those phrases are hopefully included in the .DEPARTURE_CITY grammar.

Coverage test

The goal of this test is to determine whether all of the phrases in the test set are included in the grammar.

The test set window contains two columns labelled Test Set and Saved Result(s). The first column shows the input phrases used to test the grammar. The second column shows the interpretations produced by a previous run of an interpretation test, possibly with a slightly different grammar.

1. Select Interpret All Sentences from the test set window context menu (right-click anywhere in the test set window to display its context menu). A dialog box pops up. The default options are fine, so click "OK." A third column labelled Current Result(s) should appear.
2. The new column shows the NL interpretation results of the latest run of the test set against the .DEPARTURE_CITY grammar. Because the grammar description has not changed yet, the current results and the saved results are identical.

   Notice that the phrase "we are leaving from las vegas" has the corresponding entry "<Out of Grammar>" in the Current Result(s) column. The reason is that "las vegas" is missing from the grammar. To fix this problem, add "las vegas" to the grammar.
3. Open the grammar file by double-clicking on *departure_city.grammar* in the project window.
4. Look for the definition of the City subgrammar, and insert the string "(las vegas) {return("las_vegas")}" in between "washington" and the closing square-bracket "]."
5. Save the changes and recompile the project.
6. Rerun the command Interpret All Sentences. Compare the entries in the Saved Results and the Current Results columns for the phrase "we are leaving from las vegas" (the last one on the list). Notice that the Current Results column now contains the interpretation "las_vegas" in the origin slot.

7 Right-click in the Test Set window, select “Filter”, and then select “Show Differences” from the menu. This shows only the changes from the previous version.

NL interpretation test

The goal of this test is to make sure that all of the test set phrases have the correct interpretation.

1 In the test set window, verify that each city in the Test Set column matches the corresponding interpretation in the Current Results column.

   Inspection shows that the interpretation for “i want to go from phoenix” is wrong. It reads {<origin tucson>}, but it should say {<origin phoenix>}.

2 Open the grammar file, find the bug, and fix it.

3 Recompile, and rerun the test.

Ambiguity test

The goal of this test is to find phrases in the grammar that have more than one interpretation.

Sometimes, ambiguity is an intentional part of the grammar design. For example, “portland” might refer to either Portland, Maine or Portland, Oregon. It’s up to the application to determine which one the user really means. However, ambiguity often arises unintentionally due to grammar errors.

To obtain an unambiguous result, the application must either ask the caller additional questions—“Would you like Portland Maine or Portland Oregon?”—or resolve the ambiguity by some heuristic rule—selecting the largest city, for example.

1 Select Generate Sentences from the Tools menu.

   You can also invoke the Generate Sentences command by clicking the “generate” icon in the Grammar toolbar or by pressing the F8 function key.

2 In the Generate Options dialog:

   - Enter 200 as the number of sentences to generate
   - Check the NL box and the Ambiguous Only box
   - Click OK

   You should now see phrases like “I want to go from portland” with two interpretations, namely {<origin portland_oregon>} and {<origin portland_maine>}. This is fine because the ambiguity here is intentional. There really are two cities called Portland—one in Maine and one in Oregon.

You should also see phrases like “I’m leaving from san diego” with two interpretations—{<origin san_diego>} and {<origin san_francisco>}. The latter one is erroneous.

3 Find the error in the grammar file (search for “san diego”), fix it, recompile, and rerun the Generate Sentence command.

In more complex grammars, this sort of error may be hard to find and fix, so you may want to proceed as follows:

- To locate phrases with more than one interpretation, use a filter. Inspect the submenus under “Filter” in the context menu of a test set window.
- To locate the source of the problem with a given phrase within a grammar, it is helpful to know how that phrase is parsed by the grammar. To do this, highlight the phrase in a test set window (right click over it) and select “Show Parse Tree.” A new window shows the parse tree of that phrase.

Over-generation test

The goal of this test is to find any unnecessary or unexpected phrases in the grammar.

Reviewing a list of randomly generated phrases can sometimes detect over-generation errors—unexpected and unnecessary phrases parsed by the grammar. Detecting such phrases helps recognition speed and accuracy, when those phrases are removed from the grammar.

More importantly, a “tight” grammar—one without lots of overgeneration—is usually easier to maintain. The process of getting a “tight” grammar is the equivalent to cleaning up C++ code to get a compilation with no warning messages. Your compiled code may work fine, even with the warning messages, but it will be harder to detect errors in the future.

1 Select Generate Sentences.

2 In the Generate Options dialog, enter 200 as the number of sentences to generate and then click OK.

The output of this command appears in the form of a test set with 200 random phrases. Look for any phrases that do not make sense or that would never be spoken by a caller giving departure city information.

To shorten and order the generated list, use the “Sort & Unique” filter.

Notice that the randomly generated set contains odd sentences like “i am leaving from pittsburgh please” and “we are leaving from buffalo please”. This isn’t a big problem, but most people would probably not say “please” as in these examples, so try to remove these phrases from the grammar.

Note: If you prevent unusual phrases from getting into your grammar—say once per iteration cycle— you'll end up with a nice tight grammar. Conversely, if you don't pay attention to this issue, you will most likely end up with a grammar that over-generates excessively, which can lead to poor speed and accuracy recognition rates.

3 Open the grammar file, and try to find where the word "please" occurs in the definition of .DEPARTURE_CITY. Try to change the grammar so that "please" only occurs where it sounds natural.

4 Recompile and retest to see if you have solved the problem.

For a relatively simple grammar—one generating a total of up to a few thousand phrases—the "exhaustive generation" option is recommended. This generates *every* phrase parsed by the grammar. Just check "Exhaustive" on the Generate Options dialog box.

Caution: The exhaustive option is not practical for grammars that parse more than a few thousand phrases.

Regression tests

Whenever you change a grammar, it is important to test the new version of the grammar to ensure that no errors have been introduced. If you run an old test set against a new grammar, you notice if any errors have been introduced to the previously working portions of the grammar. This is the basic paradigm of regression tests.

Whether you are testing a new grammar or doing a regression test, the steps are quite similar. The only difference is that in a regression test, you compare the results to a previous version, but when you test a new grammar, there is no previous version to compare against. The grammar tests described in previous sections may be used to test new grammars or as regression tests.

Command-line tools for grammar testing

The Nuance System toolkit includes several utility programs you can use to test your grammars. This section briefly describes the use and functionality of the following programs:

- *parse-tool*
- *generate*
- *nl-tool*

- *Xapp*
- *batchrec*

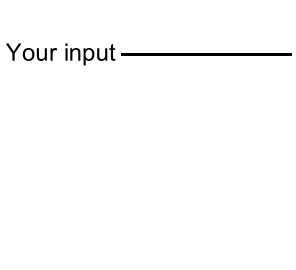
See the online documentation for complete reference information on these utilities.

parse-tool

parse-tool assists you in testing whether your grammar parses a given sentence correctly. *parse-tool* takes sentences from standard input and returns a value indicating whether the sentence was successfully parsed. Use the option *-print_trees* to get a display of the grammar paths traversed to match the utterance. For example:

```
> parse-tool -package numbers -print-trees
```

produces the following output:




```
Ready
thirty two
Sentence: "thirty two"
.N0-99
    DECADE
        thirty
    NZDIGIT
        two
```

generate

generate outputs the sentences defined by paths in your grammars. The generation scheme could be exhaustive or random. This tool is typically used to test for overgeneration—that is, to detect nonsensical or inappropriate sentences that your grammar supports.

To run *generate*, specify the package and the name of the grammar (top-level or subgrammar) within the package that you want to analyze.

By default, *generate* continues randomly generating possible sentences until you stop the program. Use the *-num* option to specify the number of sentences to generate, as in the following command:

```
> generate -package numbers -grammar -N0-99 -num 5
```

Then the output might look like:

```
eighty one
seven one
fifty
```

```
fourteen
thirty two
```

If this program generates unwanted or unexpected sentences, your grammar is overgenerating.

To print out the interpretations for each generated sentence, specify the *-nl* option. For example:

```
> generate -package numbers -grammar -N0-99 -num 5 -nl
```

To test your grammars for ambiguity, use the option *-ambig*. This option causes *generate* to output only sentences defined by the grammar that have more than one interpretation. When using the *-ambig* option, the grammar you specify *must* be a top-level grammar, as in the following:

```
> generate -package myPackName -grammar .myTopLevelGrammar -ambig
```

nl-tool

nl-tool takes sentences from standard input and outputs the interpretations for them. *nl-tool* requires the specification of a package and a top-level grammar within that package to use for recognition, as illustrated in the following command line:

```
> nl-tool -package myPackageName -grammar .Sentence
```

nl-tool also lets you perform batch mode analysis of interpretations using the arguments *-gen_ref* and *-compare_to*.

Xapp

Xapp is a graphical tool that performs recognition of audio utterances by supplying a transcription of each recognized phrase and the confidence score of the recognition.

This tool allows you to run a simple real-time test on a grammar by speaking utterances that you expect to be parsed by your grammar, and then checking the result.

To run *Xapp*:

1 Start a *RecServer* from a command line window by typing a command like the following:

```
> recserver -package my-package lm.Addresses=my-server-machine
```

where *my-package* is a compiled package name, and *my-server-machine* is the name of the host machine where the Nuance Licence Manager (*nlm* process) is running.

The Nuance System also lets you load and unload recognition packages from a recognition server without restarting the server. For more details on this feature, see the *Nuance Application Developer's Guide*.

The *RecServer* is completely initialized when it displays the message:

```
Recserver ready to accept connections on port XXXX
```

2 In another window, invoke *Xapp* with the following command:

```
> xapp -package my-package lm.Addresses=my-server-machine
```

where *my-package* and *my-server-machine* are identical to the strings used for the command in Step 1. On the Windows platform you can find the *Xapp* command in the Nuance start menu.

3 Select the grammar you want to test.

4 Cycle through the following steps until you are done:

- Click the **Listen** button.
- Speak a phrase that you expect to be parsed by your grammar.
- Check out the result returned by *Xapp* in the "Recognized Speech" area of the window. No interpretation is displayed if your grammar does not have any natural language commands.

Note: To perform similar test using a text-based interface, use *RCApp*.

batchrec

An important tool that you can use to test your grammar is *batchrec*. This tool is rather sophisticated as it requires a set of pre-recorded audio data—usually gathered from field data or from a special data collection.

Basically, you run the spoken data through the *batchrec* and it evaluates recognition accuracy scores for each utterance in your data set. See Chapter 6 for more information on *batchrec*.

Chapter 6

Testing recognition performance

6

This chapter describes how to test recognition performance of an application using the command-line program *batchrec*. The *batchrec* program performs recognition on a set of recorded audio files. If you also provide the correct results (either utterance transcriptions or natural language interpretations, or both) for the utterances in the audio files, *batchrec* scores the result for each file and prints cumulative accuracy statistics.

You should use *batchrec* to analyze and tune the performance of your live applications. Recognizing pre-recorded utterances instead of live speech lets you both process a lot of data quickly and process the same data multiple times. This enables you to:

- Establish a baseline test for recognizer accuracy
- Measure the speed of the recognizer
- Estimate performance on a live task, in advance
- Tune the recognizer configuration by varying parameter values while holding the speech data constant

You can record digital audio files for use with *batchrec* via a running application, by using the Nuance waveform editor *Xwavedit*, or by using third-party recording software. Nuance software records files in either the SPHERE-headered *.wav* format or the RIFF format commonly used on Windows NT. The *batchrec* tool can process either of these formats. You can also use the *wavconvert* tool to convert files between the *.wav*, RIFF, and *.au* formats.

Note: See the online documentation for information on the *wavconvert* and *Xwavedit* utilities.

Choosing *batchrec* test sets

The *batchrec* program provides you with a statistical measure of recognition accuracy for a set of audio files. To get useful results, you must test the right set of recorded utterances, and interpret the resulting statistics carefully.

The utterances in your *batchrec* test set should be as representative as possible of the utterances that your application will recognize when deployed. Characteristics you should consider include:

- Gender and accent of speaker
- Content of speech
- Rate (speed) of speech
- Method of audio capture
- Ambient noise conditions

In addition, there is considerable variation among speakers, and even among different utterances by the same speaker. Therefore, for statistically reliable results you must provide a large sample set. To accurately measure speaker-independent recognition accuracy, your sample set should include at least 500 utterances spoken by at least 20 different people whose speech characteristics mirror the target speaker population.

Note: The recognition system uses several adaptive parameters to compensate for variation across audio channels and to estimate the background/channel noise in the signal. This means that each recognition result is somewhat dependent on the preceding utterances. At the start of the first utterance there is no information about the signal, so the system uses a set of default values. If the signal is very noisy or very clean these default values are not good, so the system makes an estimate of the signal noise and stores it for the next utterance to use. All subsequent utterances can improve this estimate, so the value for a particular utterance depends on all utterances before it. The effect on recognition is usually negligible, but if you change the order of your test utterances, the recognition score and perhaps the result may change. The recognizer always starts in the same state, so if you do not change the order of the utterances, your test set always produces the same results.

Using *batchrec*

To invoke *batchrec*, you must specify, at a minimum:

- The recognition package

- A file containing a list of the files to be processed

For example:

```
> batchrec -package package_name -testset testset_list
```

batchrec also supports a number of additional arguments, described in "Optional arguments" on page 78. The following section describes the format of the file list passed to *batchrec*.

Creating the testset file

The file you specify with the *-testset* option lists the digital audio files to be recognized, one audio file per line. Each audio file is processed as a single utterance for output and scoring purposes, regardless of its content.

The testset file also identifies the grammar to use to recognize each audio file. Many recognition packages contain several top-level grammars, to be used for different recognition tasks. Use the **Grammar* keyword to identify the grammar to use to recognize particular files. Typically, you create command blocks, listing the set of files to recognize with each grammar. For example:

```
*Grammar .YesNo
/home/usr1/waveforms/yes.wav
/home/usr1/waveforms/no.wav
*Grammar .StockQuote
/home/usr1/waveforms/stocks001.wav
/home/usr1/waveforms/stocks002.wav
/home/usr1/waveforms/stocks003.wav
```

With this test set, the files *yes.wav* and *no.wav* are recognized using the *.YesNo* grammar, while the files *stocks001.wav*, *stocks002.wav*, and *stocks003.wav* are recognized using the *.StockQuote* grammar.

Note: If the recognition package you specify on the *batchrec* command line contains only one top-level grammar, this grammar is used by default. However, it is good practice to always specify the grammar on the first line of your testset file. You can also set a default grammar for a *batchrec* run with the argument *-grammar*.

Additional commands

In addition to the **Grammar* command, *batchrec* supports a number of additional commands you can include in your testset file:

`*Echo` *text*

Lets you specify text to be written out to the screen during processing.

`*Exit`

Lets you explicitly exit the testset at a given point.

*`SetParam` *param_name=param_value*

Sets the specified Nuance parameter to the given value. The parameter must be runtime-settable. For information about Nuance parameters see the online documentation or the *Nuance Application Developer's Guide*.

`*GetParam` *param_name*

Prints out the current value of the specified parameter.

`*NewAudioChannel`

Indicates that the following audio files were generated on a different audio channel than the previous file. Clears inserted dynamic grammars with `CALL` persistence.

You can also include commands for working with dynamic grammars and for performing speaker verification. Dynamic grammar commands are described in the next section. For information on *batchrec* speaker verification commands, see the *Nuance Verifier Developer's Guide*.

Dynamic grammar commands

You can use *batchrec* to evaluate the performance of applications that include dynamic grammar functionality. This includes both recognizing against dynamic grammars, and using the enrollment facility to add a new voice-trained phrase to a dynamic grammar (this requires that you provide a transcriptions file when you start *batchrec*). The related *batchrec* commands are described here. See "Dynamic grammars" on page 27 for information on the Nuance System's dynamic grammar functionality. In the references below, the database handle (such as *db_handle*) must be a positive integer; zero is not allowed as a database handle value.

`*OpenDynamicGrammarDatabase` *db_handle DBDescriptor_arguments*

Creates a connection to the dynamic grammar database containing the dynamic grammars you want to use for testing. This must be a database where *-dbclass* is equal to *dgdb*. See the *Nuance Application Developer's Guide* for information on the arguments required for a given database provider.

`*CloseDynamicGrammarDatabase` *db_handle*

Closes the connection to the specified database.

`*NewDynamicGrammarEmpty` *db_handle db_key ok_to_overwrite*

Creates a new, empty record in a dynamic grammar database.

`*NewDynamicGrammarFromGSL` *db_handle db_key gsl_file ok_to_overwrite*

Creates a new record in a dynamic grammar database containing the given GSL expressions.

`*NewDynamicGrammarFromPhraseList` *db_handle db_key pl_file ok_to_overwrite*

Creates a new record in a dynamic grammar database containing the specified set of phrases. The phrase list file (*pl_file*) contains one phrase per line, in the format:

phrase_id phrase_text nl_command probability

For example:

```
user_112 "john doe" "{<user john_smith>}" 1
```

Note that the natural language command *must* be enclosed in double quotes.

`*DeleteDynamicGrammar` *db_handle db_key*

Removes a record from a dynamic grammar database.

`*CopyDynamicGrammar` *source_db_handle source_db_key target_db_handle target_db_key ok_to_overwrite*

Creates a new record in a dynamic grammar database, containing the contents of an existing record.

`*AddPhraseToDynamicGrammar` *db_handle db_key p_id p_text p_nl*
`*AddPhraseList` *db_handle db_key pl_file*

Adds a single phrase or a set of phrases to a dynamic grammar.

`*RemovePhrase` *db_handle db_key p_id*

Removes a phrase from a dynamic grammar.

`*QueryDynamicGrammarExists` *db_handle db_key*

Prints out whether the given database contains a dynamic grammar at a given key.

`*QueryDynamicGrammarContents` *db_handle db_key*
`*QueryDynamicGrammarContentsWithID` *db_handle db_key phrase_id*

Prints out the contents of a specified dynamic grammar database record, or of the phrases in that record with a given ID.

`*InsertDynamicGrammar` *db_handle db_key label* `{CALL | PERMANENT}`

Inserts a dynamic grammar from a database into a static grammar at the given label, with the specified persistence (`CALL` or `PERMANENT`). CALL persistence means that the insertion remains until next `NewAudioChannel` line.

`*EnrollNewPhrase` *grammar db_handle db_key* {`NEEDED` | `ALL`}

Begins a dynamic grammar enrollment session. The enrolled phrase is added to the specified dynamic grammar. Note that to be able to perform enrollment, you must provide either a transcription or natural language transcription file when you start *batchrec*, using the `-transcriptions` or `-nl_transcriptions` options. The format of these enrollment transcription files is the same as the format of the transcription files used for recognition, except for the use of noise markers, which are not allowed in enrollment transcription files. See "Optional arguments" on page 78 for a description.

For the final argument, specify `NEEDED` to have the new phrase added to the grammar as soon as enough consistent enrollments are found. Specify `ALL` if you want all enrollments to be used as listed in the testset file, before the phrase is added to the grammar.

`*EnrollCommitPhrase`

Commits an updated dynamic grammar to the database.

`*EnrollAbortPhrase`

Ends an enrollment session without updating the dynamic grammar.

`*ModifyPhrase` *db_handle db_key p_id p_text p_nl*

Modifies a voice-enrolled phrase in a dynamic grammar.

`*CompileAndInsertDynamicGrammar` *gsl_file label persistence* {`CALL` | `PERMANENT`}

Compiles a dynamic grammar from a GSL file and inserts it into a static grammar at the given label, with the specified persistence (`CALL` or `PERMANENT`).

Optional arguments

batchrec supports a number of optional arguments:

`-transcriptions` *transcriptions_file*

If you supply a file containing a transcription for the utterance in each audio file in the test set, *batchrec* prints accuracy statistics, both per sentence and cumulative. Each line in the transcriptions file contains a file name followed by a transcription string. The file name can be as short as the base name of the audio file, or as long as the complete path of the audio file. Supply as much of the path as is necessary to uniquely identify all files. If all files in the test set are found in one directory and have unique base names, only the

base name needs to be supplied. Everything after the file name is the transcription of the audio file. For example:

```
datadir1/file3.wav phoenix arizona
datadir1/file1.wav chicago illinois
datadir2/file3.wav boston massachusetts
```

Transcriptions can be listed in any order. Words in the transcription must be spelled exactly as they appear in the grammar, including letter case. If digits are spelled out in the grammar ("four" instead of "4"), they must be spelled out in the transcriptions file, too.

`-nl_transcriptions` *nl_transcriptions_file*

If your grammar includes natural language interpretations, you can supply a file with transcriptions of the correct natural language interpretation for each audio file, and *batchrec* prints accuracy statistics for the natural language interpretations returned by the recognizer. Each line in the file contains a file name and then a natural language interpretation. Everything after the file name is the natural language interpretation for that audio file. You can also generate the NL transcriptions file from a standard transcriptions file using the *generate-nlref* tool.

You can list natural language transcriptions in any order. A natural language interpretation is a set of named slots, with a value for each. Each slot/value pair is enclosed in angle brackets. Inside the angle brackets, the first word is the slot name and everything after the first space is the slot value—so slot names are limited to a single word, while slot values may be one or more words. If the slot name or value includes an angle bracket, the entire slot name or value must be enclosed in double quotes. Slot and value names must be spelled exactly as they appear in the grammar. For more information on how to specify a natural language component in your grammar and how to use the natural language recognition result, see "Defining natural language interpretations" on page 46. An example transcriptions file is:

```
datadir1/file3.wav <city phoenix> <state arizona>
datadir1/file1.wav <city chicago> <state illinois>
datadir2/file3.wav <city boston> <state massachusetts>
```

Note: A common scoring problem that arises with digit grammars is that the natural language system produces a string value, while the transcription gives an integer value. To force a sequence of digits to be treated as a string, enclose it in double quotes. The transcription file would look like this:

```
datadir1/file1.wav <digits "123456">
```

rather than like this:

```
datadir1/file1.wav <digits 123456>
```

If N-best recognition is turned on, *batchrec* also prints N-best accuracy statistics, both for recognition accuracy and natural language accuracy. The

N-best accuracy statistics are just like the regular statistics except that they measure the accuracy you would achieve if the system could automatically choose the best item out of the N-best list. For more information on N-best recognition, see the *Nuance Application Developer's Guide*.

`-grammar` *grammar_name*

Sets a default grammar to be used if none is specified with a `*Grammar` command.

`-vrs`

This option causes *batchrec* to run in a client/server environment by causing the program to connect to a running recognition server or resource manager instead of performing recognition in the same process.

`-rcengine`

This option causes *batchrec* to run in a client/server environment, connecting to a recognition server process as a client using the RCEngine interface.

`-rcapi`

This option causes *batchrec* to run in a client/server environment, connecting to a recognition server process as a RecClient using the RCAPI. The audio simulator is used as audio source be default. If you want to have audio data come from a telephone line, you must specify the following parameters and command-line options:

```
audio.Provider=dialogic
audio.Device=<telephony port #1>
-lineB_device <telephony port #2>
-lineB_number <phone number for port #2>
```

batchrec then spins off a thread taking phone calls at port #2, playing files from port #2, and doing recognition in the main thread which is getting audio data from port #1.

`-rc_timeout` *float*

When the *-rcapi* option is used, you can specify this option to indicate a recognition timeout limit. This allows *batchrec* to simulate timeout limits and assess the effects of different recognizer timeout settings. The default is 100 seconds.

`-app`

This option causes *batchrec* to run in a client/server environment, connecting to a recognition server process as a client using the Dialog Builder. The audio simulator is used as audio source be default. If you want to have audio data

come from a telephone line, the following parameters and command-line options must be specified:

```
audio.Provider=dialogic
audio.Device=<telephony port #1>
-lineB_device <telephony port #2>
-lineB_number <phone number for port #2>
```

batchrec then spins off a thread taking phone calls at port #2, playing files from port #2, and doing recognition in the main thread which is getting audio data from port #1.

`-endpoint | -dont_endpoint`

The *-endpoint* and *-dont_endpoint* options specify whether or not endpointing is applied to the speech samples before recognition.
The default is *-dont_endpoint* because it is assumed that the recorded utterances have already been endpointed. If *-endpoint* is specified, then leading and trailing silence is stripped out of the speech sample before it is handed to the recognizer. Nuance recommends that you do not endpoint speech files that were recorded, and therefore already endpointed, by a Nuance application. Results differ, depending on how this option is set.

`-print_confidence_scores`

The recognition accuracy at various confidence thresholds is output for both transcriptions and natural language transcriptions. To obtain maximum output set the parameter `rec.ConfidenceRejectionThreshold` to 0 or -1 on *batchrec's* command line.

`-print_word_confidence_scores`

This option causes *batchrec* to include the confidence score for each word in each utterance in its output.

`-debug_level` *0-5*
`-debug_level_during_init` *0-5*

You can use *-debug_level* and *-debug_level_during_init* to control how much information is printed about the processing. They take a value from 0 to 5; 0 suppresses most output, while 5 prints the largest amount of information. The default level for both of the settings is 2.

Setting Nuance parameters

You can also specify Nuance parameters on the *batchrec* command line. For example:

```
> batchrec -package package_directory -testset testset_list_file
  -transcriptions transcriptions_file -endpoint rec.DoNBest=TRUE
```

causes *batchrec* to perform N-best processing.

See the online documentation or the *Nuance Application Developer's Guide* for more information on Nuance parameters.

Output from *batchrec*

The following shows sample *batchrec* output for a single audio file, where *batchrec* was run with both a transcriptions and an NL transcriptions file:

```
File 72:       /home/tests/myapp/testset/numfood171.wav
Grammar:       .Sentence
Transcription: buy him seven fish now
NL Transcript: <get-how buy> <for him> <count seven> <food fish>
               <when now>
Result #0:     buy him seven fish now (conf: 66, NL conf: 63)
NL Res.#0:     <when "now"> <count "seven"> <for "him"> <get-how "buy">
               <food "fish">
Total Audio:   2.63 sec
Utt. Times:    0.37 secs 0.141xRT (0.37 usr 000 sys 0.141xcpuRT) 100%cpu
Avg. Time:     0.34 secs 0.132xRT (0.33 usr 0.00 sys 0.130xcpuRT) 99%cpu
Rec Errors:    0 ins, 0 del, 0 sub = 0.00% of 5 words.
Rec Total:     0 ins, 0 del, 2 sub = 0.56% of 360 words (2.78% of 72 files).
NL Status:     correct
NL Total:      0 rejects, 2 incorrect = 2.78% error on 72 files
```

This output includes the following information:

`File 72: /home/tests/myapp/testset/numfood171.wav`

Provides the name and location of the audio file being analyzed. In this case, this is the seventy-second file recognized by this *batchrec* run.

`Grammar: .Sentence`

Names the recognition grammar used to recognize this file.

`Transcription: buy him seven fish now`

Prints the transcription of the utterance in the audio file, as provided in the file specified with the `-transcriptions` option.

`NL Transcript: <get-how buy> <for him> <count seven> <food fish> <when now>`

Prints the natural language transcription for the utterance, as provided in the file specified with the `-nl_transcriptions` option.

```
Result #0:  buy him seven fish now (conf: 66, NL conf: 63)
```

Prints the recognition result returned by the recognizer and the confidence scores of that result.

```
NL Res.#0:  <when "now"> <count "seven"> <for "him"> <get-how
"buy"> <food "fish">
```

Prints the natural language interpretation generated from the recognition result.

Note: In the previous example, only one result is returned. If N-best processing is enabled (by setting the parameter *rec.DoNBest* to *TRUE*), multiple recognition and natural language results may be returned.

```
Total Audio: 2.63 sec
```

Provides the length of the recording.

```
Utt. Times: 0.37 secs 0.141xRT (0.37 usr 000 sys 0.141xcpuRT)
100%cpu
```

Provides the wall clock and CPU times used to process the file, followed by the percentage of CPU time dedicated to recognition processing during this period. This file, for example, took 0.37 seconds to process, and the recognition process got 100% of the CPU's time during processing.

```
Avg. Time: 0.34 secs 0.132xRT (0.33 usr 0.00 sys 0.130xcpuRT)
99%cpu
```

Provides the average wall clock and CPU times used to process files to this point, and the percentage of CPU time dedicated to recognition processing. The `xcpuRT` value is typically the most useful measure of speed.

```
Rec Errors: 0 ins, 0 del, 0 sub = 0.00% of 5 words
```

Prints error statistics for this file, including number of incorrect words inserted by the recognizer, number of words deleted, and number of words misrecognized. This line also includes the word count and the per-word error rate.

```
Rec Total: 0 ins, 0 del, 2 sub = 0.56% of 360 words (2.78% of 72
files)
```

Prints error statistics for all files processed so far, including per-word and per-file error rates. In this example, for the 72 files processed so far, there have been no insertions, no deletions, and two substitutions (misrecognitions). 0.56% of words have been misrecognized and 2.78% of the files have yielded a recognition error.

```
NL Status: correct
```

Indicates whether the natural language interpretation was correct, incorrect, or rejected.

```
NL Total: 0 rejects, 2 incorrect = 2.78% error on 72 files
```

Provides natural language error statistics for all files processed so far, including the number of rejects, the number of incorrect interpretations produced, and the per-file error rate. In this example, 2.78% of the files have yielded a recognition error. This error rate is the same as the recognition error rate for this example. This can occur when the grammar produces a one-to-one word to slot mapping.

Chapter 7

Creating application-specific dictionaries

7

The Nuance System represents each word in a recognition vocabulary as a sequence of phonetic symbols. These sequences provide the mapping between a word in the vocabulary and its pronunciation, and reside in dictionary files. The Nuance System dictionary provides phonetic pronunciations for more than 100,000 English words. Ideally, all the words you need for your application will be part of the Nuance dictionary file. However, some applications may require words or alternative pronunciations, such as proper names or special acronyms, that the Nuance System dictionary does not contain. In these situations, you can define additional word pronunciations in a separate, auxiliary dictionary file.

When you compile your grammars, the compilation process will tell you whether your grammars contain any words that are not part of the main Nuance System dictionary by generating an error message. To resolve this type of compilation error, you provide pronunciations for those words not listed in the Nuance dictionary, as described in “Adding missing pronunciations” on page 35, either by using the automatic pronunciation generator or by providing a dictionary file with the missing pronunciations.

Note: Remember that the automatic pronunciation generator provides a fast, easy mechanism for generating pronunciations during application development. However, pronunciations that are created manually are typically more accurate, assuming they are carefully created and tested.

The Nuance phoneme set

The Nuance System uses a a set of symbols to represent *phonemes*. Roughly speaking, there is one phoneme corresponding to each spoken sound in a given language. For spoken English, for example, there are more than 40 different phonemes. Because letters can be pronounced differently in different words and because different letters can represent the same sound, there is not a one-to-one

correspondence between letters and phonemes. For example, the [k] sound and corresponding /k/ phoneme occur for the *c* in *cat*, the *k* in *keep*, the *ck* in *tick*, the *ch* in *echo*, and the *q* in *Iraq*. Similarly, the letter *c* can be pronounced using the /s/ phoneme as well as the /k/ phoneme (as in *circle*). Therefore, a mapping between the pronunciations of a word (specified as a phoneme sequence) and the spelling is necessary input to the recognizer.

The Nuance phoneme set is specified using the Computer Phonetic Alphabet (CPA). The CPA provides a system for easily expressing the phonemes in notation defined by the IPA (International Phonetic Alphabet) using a standard keyboard. The following table lists the phonemes used by the Nuance System to express pronunciations for American English:

Table 8: Mappings for American English

Phonetic Category	Phoneme	Example	Phoneme	Example
Vowels	**i**	fl**ee**t	**u**	bl**ue**
	I	d**i**mple	**U**	b**oo**k
	e	d**a**te	**o**	sh**ow**
	E	b**e**t	**O**	c**augh**t[a]
	a	c**a**t	**A**	f**a**ther, c**o**t[a]
	aj	s**i**de	**aw**	c**ou**ch
	Oj	t**oy**	***r**	b**ir**d
	^	c**u**t	*	**a**live
Stops	**p**	**p**ut	**b**	**b**all
	t	**t**ake	**d**	**d**ice
	k	**c**atch	**g**	**g**ate
Flap	**!**	bu**tt**er, hi**dd**en		
Nasals	**m**	**m**ile	**g~**	runni**ng**
	n	**n**ap		

Table 8: Mappings for American English

Phonetic Category	Phoneme	Example	Phoneme	Example
Fricatives	**f**	***f***riend	**v**	***v***oice
	T	pa***th***	**D**	***th***em
	s	***s***it	**z**	***z***ebra
	S	***sh***ield	**Z**	vi***s***ion
	h	***h***ave		
Affricates	**tS**	***ch***ur***ch***	**dZ**	***j***u***dg***e
Approximants	**j**	***y****es*	**w**	***w***in, ***wh***ich
	r	***r****ow*	**l**	**l**ame
Others (reserved for Compiler)	**-**	(silence)		
	inh	(inhale)	**exh**	(exhale)
	clk	(click)	**rej**	(other)

a. Many American dialects do not distinguish between /A/ and /O/, and the distribution of the two sounds among specific words often varies. When adding words to your dictionary, look at similarly spelled words using the utility program *pronounce* to get an idea of which symbol is appropriate. Providing a variant with each sound is often useful.

It is important that you be very careful when using the phoneme set, because the symbols representing phonemes do not consistently correspond to the letters in the spelling of a word. Because one phoneme can be represented by several different letters or letter combinations, those same letters can represent other phonemes in other words. The following table shows some typical mappings from English letters to phonemes.

Table 9: Mapping English-language letters

Letter	Phonemes	Example
a	**e**	***a***ble
	A	f***a***ther
	a	***a***pple, p***a***n
	O	b***a***ll, c***augh***t, p***aw***n
	*	***a***live, zebr***a*** (unstressed vowels)
b	**b**	***b***all
c	**k**	***c***at, e***ch***o
	tS	***ch***ild
d	**d**	***d***ogs
	!	bri***d***le
	t	aske***d***, kisse***d***
e	**i**	***e***ven
	E	***e***cho
	u	n***ew***
	*	ag***e***nt
f	**f**	***f***ather
g	**g**	***g***olf
	dZ	***g***iant
h	**h**	***h***otel
i	**aj**	***i***vory, t***i***me
	I	***I***ndia
	i	L***i***sa, sh***ie***ld
	*	san***i***ty, aph***i***d
j	**dZ**	***j***ump
k	**k**	***k***ey

Table 9: Mapping English-language letters

Letter	Phonemes	Example
l	**l**	***l***aw
	*** l**	ab***le***
m	**m**	***M***ike, thu***mb***
	*** m**	Ad***am***, wisd***om***
n	**n**	***n***ew, ***kn***ot
	*** n**	wid***en***, nati***on***
n+g	**g~**	ri***ng***
o	**o**	***o***pen, c***oa***t, sh***ow***
	A	***o***live
	aw	c***ow***
	O	b***ough***t
	u	sh***oe***, l***oo***p
	U	b***oo***k, w***ou***ld
	^	c***o***mpany
	*****	carr***o***t
	Oj	t***oy***
p	**p**	***p***ull
	f	a***ph***id
q	**k w**	***qu***ick
	k	Ira***q***
r	**r**	***r***ing
	***r**	b***ir***d, rid***er***, gramm***ar***
s	**s**	***s***oup
	S	***sh***ell
	z	dog***s***, wi***s***dom
	Z	vi***si***on