NUANCE

Nuance Speech Recognition System
Version 8.5

Grammar Developer's Guide

Nuance Speech Recognition System 8.5
Grammar Developer's Guide

Contents

About this guide

This guide provides guidelines for developing and testing a grammar and describes the tools that Nuance offers for building efficient and accurate recognition packages.

Audience

This guide is intended for developers wanting information about the major tasks involved in building a recognition package, including:

- Understanding the grammar development process
- Designing and writing a grammar
- Building a recognition package
- Supporting natural language interpretation
- Testing a grammar
- Defining application-specific word pronunciations
- Using the Analysis and Tuning Tools to analyze the performance of an application and run tuning experiments

Organization

This guide is organized as follows:

Chapter 1 gives general guidelines for developing complex grammars and outlines the grammar development process.

Chapter 2 explains the basic features of the Grammar Specification Language used to describe a grammar, and describes grammar hierarchies and grammar probabilities.

Chapter 3 introduces dynamic grammars created using API functions, and describes how to refer to dynamic grammars using external rule references and how to use just-in-time grammars. It also provides information on configuring your system to use external rule references and just-in-time grammars

Chapter 4 defines natural language interpretations and explains how to add them to a grammar. This chapter also introduces advanced interpretation features like functions and data structures.

Chapter 5 introduces the Say Anything™ feature and describes how to write statistical language models (SLM) grammars. It also explains how to use the robust natural language interpretation (robust NL) technology.

Chapter 6 describes how to compile a grammar into a recognition package and specify additional word pronunciations. It also discusses some optimization and performance issues.

Chapter 7 presents a comprehensive way of testing all aspects of a grammar, including interpretation and ambiguity, as well as performing regression tests.

Chapter 8 describes how to use the Nuance *batchrec* utility to test the performance of recognition packages.

Chapter 9 describes the Analysis and Tuning Tools, a set of command-line tools to analyze the performance of an application and run tuning experiments.

Chapter 10 explains how to make a dictionary file for an application, and how to specify word pronunciations using the Computer Phonetic Alphabet phoneme set.

Appendix A contains a formal definition of the Grammar Specification Language syntax and a summary of all the files used by the grammar compiler.

Appendix B provides advanced information on the Nuance SLM utilities.

Appendix C describes how Nuance supports W3C grammars. It also lists the Nuance extensions to the W3C format and the natural language features added to it.

Appendix D provides the DTDs for the grammar overrides file and the scores file.

Appendix E describes how to provide transcriptions that follow the standard Nuance transcriptions conventions.

Related documentation

The Nuance System documentation contains a set of developer guides as well as comprehensive online reference documentation. See *Introduction to the Nuance System* for the list of available documents.

Typographical conventions

Nuance manuals use the following text conventions:

italic text	Indicates variables, file and path names, program names, program options, web and email addresses, as well as terms introduced for the first time. For example: Edit the *ag.cfg* configuration file.
`Courier New`	Indicates method/member functions, parameter names and values, program constants, and onscreen program output. For example: Nuance recommends that you set the parameter `audio.OutputVolume` to `255`.
`> Courier New`	Indicates commands or characters you type in and the responses that appear on the screen. The > character indicates the MS-DOS command prompt or Unix shell. Everything after this character is intended to be typed in. For example: `> resource-manager` In this example, the text that you actually type at the command line is "resource-manager"
`Courier New`	Indicates a value that you replace. For example: The usage for the *nlm* utility is: `> nlm` *`license_key`* In this example, *`license_key`* is a value that you replace, so the text you actually type could be, for example: > nlm `ncr8-16-100-a-b22-333c4d55eeff`

Note: The Nuance System runs on both Windows and Unix platforms. Windows syntax is typically used throughout the documentation. If you are running on a Unix platform, substitute Unix syntax. For example, use *$NUANCE* wherever *%NUANCE%* appears, and use "/" in place of "\" in path names. Differences in usage or functionality on Windows and Unix platforms are noted where relevant.

Where to get help

If you have questions or problems, technical support is provided through Nuance Technical Support Online, a web-based resource center that includes technical support guides on specific topics, access to software updates, and technical support news. Access to Nuance Technical Support Online is available to members of our partner programs and direct customers of Nuance. If you are a member, go to *support.nuance.com* to log on.

To submit comments on the documentation, please send email directly to *techdoc@nuance.com*. Note that no technical support is provided through this email address. Technical support is provided through the Nuance Technical Support Online.

Chapter 1

The grammar development process

1

An intuitive user interface that guides the user through a constrained but purposeful interaction is a crucial component of speech recognition application design. The user speaks naturally, yet, well-designed prompts ensure that the user′s utterances fall largely within an expected set of phrases, and well-designed grammars cover these expected phrases and allow the recognizer to achieve a high accuracy rate.

This chapter describes an approach to grammar development designed to help you create effective grammars. These guidelines are the result of experience gained by developers writing grammars for real-world applications. They provide a tested approach to the complex task of designing and writing an effective grammar—a critical component of a good speech recognition application.

Note: The objective of this chapter is to introduce the grammar development process; the most common grammar format—Nuance Grammar Specification Language—is used for the examples.

Principles of effective grammar design

The following sections describe general principles that Nuance advocates for the design of quality grammars. Keep these guidelines in mind when writing grammars for your particular application.

Programming versus grammar writing

The complexity of a grammar greatly affects the speed and accuracy of the recognizer. Complex grammars must be constructed with as much care as complex software programs.

Grammar writing is an unfamiliar task for most software developers, and creating a high-quality, error-free grammar requires somewhat different skills than programming in a language like Java or C++. Grammars are inherently non-procedural and thus software programming and grammar writing cannot be approached in the same way.

Conceptually, a grammar is a set of phrases—possibly infinite—that a caller is expected to say during a dialog in response to a particular prompt. The grammar writer's task is to *predict* that set of phrases and *encode* them in the grammar.

Predicting how callers will speak

Of the two tasks, predicting and encoding, predicting the set of responses is by far the more difficult. Even if you are just expecting a simple yes/no response, you will likely get a wide range of responses from real callers, such as "yeah," "yup," "no way," "correct," and others that you might not guess in advance.

Grammar writing is an iterative process: you make your best guess, collect some real data, refine the grammar, get some more data, refine further, and so on. As you refine the grammar by adding and removing phrases, it more closely approximates the way callers speak to the application.

In practice, you are not able to include *all* of the responses that can occur in your application because you cannot control how people speak. When a caller says a word or phrase that cannot be parsed by the grammar, the word or phrase is considered *out-of-grammar.* As a rule of thumb, a 5% out-of-grammar rate is considered acceptable. Even 10-20% out-of-grammar rates are not uncommon for certain types of recognition tasks.

A good advice is to avoid putting too much effort into thinking of *every* alternative way of saying something right from the start—this will largely be a wasted effort. Instead, focus on guessing the most common ways that people will respond and build those into your first grammar release. After you have collected some data, expand your grammar based on that field data.

How do you guess the most common responses? Fortunately, it turns out that there are two types of responses that are by far the most common:

- The information item by itself
- The literal response to the question wording

So, if you ask "What is your departure city?" most responses will be just a city name like "New York" with no other verbiage. A smaller group of responses will contain phrases like "My departure city is Miami" or "departure from Miami" in direct response to the prompt wording. If you change the question to

"What city would you like?" you'll still get city-only responses, but you'll also get "I'd like Miami" responses. In this second case, you probably won't get many (or any) responses of the form "My departure city is Miami."

Therefore it is important to:

- Word prompts carefully
- Coordinate grammars and prompts, making sure that your grammars correspond closely to the prompt wording

Furthermore, whenever you change the wording of a prompt, be sure to modify the corresponding grammar as well.

Steps in the development process

The following sections describe the generic process of developing a grammar. Details on specific concepts and terminology are included later in this guide.

The tasks you should follow while developing a grammar are:

1. Define the dialog
2. Identify the information items and define the slots
3. Design the prompts
4. Determine the appropriate type of grammar to use
5. Anticipate the caller responses
6. Identify the *core* and *filler* portions of your grammars
7. Write the code for your grammars
8. Add natural language commands

Define the dialog

It is important to define the dialog before starting to write a grammar, because the dialog determines what grammars you have to write. For an important commercial application, the dialog is normally defined in a formal dialog specification document, though you can use whatever type of documentation makes sense for your project. For instance, a one-time demo would not require a large amount of detailed specification. In any case, it is important to have a good understanding of the dialog *before* you start to write the grammars.

At a minimum, you should answer the following questions:

1 What pieces of information are required to complete the task?

2 In what order will the information be requested?

3 Will the dialog request one piece of information at a time in a particular order—a *directed dialog*—or will it allow several pieces of information at once, in any order, and prompt for missing items as necessary—a *mixed-initiative dialog*?

The answers to these questions determine the shape and content of the grammars you need to develop. See the *Introduction to the Nuance System* for more information on developing dialogs.

Identify the information items and define the slots

Once your dialog has been defined, it is relatively simple to determine what items your dialog should capture. Normally, you would use one slot for each piece of information. A *slot* is similar to an identifier in a data structure in that it holds a value of a certain type. (See the introduction to Chapter 4 for more information on slots.)

For example, if you're creating an air travel application, you might need to collect two cities (origin and destination), a date, and a time, and then confirm the validity of information assembled (a yes/no question). That's five pieces of information in all. At this point, you may also want to determine the *format* and *type* in which the information will be returned.

You can summarize all that information in a table like the following:

Item	Slot name	Value format	Value type
city #1	origin	3-letter code	string
city #2	dest	3-letter code	string
date	date	[<month> <day>]	NL structure
time	time	0-2359	integer
yes/no	confirm	"yes" or "no"	string

This information helps you set up your grammars to return the right values in the right format in the right slots.

Design the prompts

After defining the dialog, information items, and slots, you are ready to write the wording for your dialog's prompts.

Prompt design is best done *before* writing the grammars because prompt wording can greatly affect the wording of the caller responses, as pointed out earlier. The grammar needs to capture those responses, so if the prompts are changing frequently while the grammars are developed, you will probably have to do a lot of rework.

Note: Core items, such as city and name lists, can typically be developed earlier in the process—that is, before completing the dialog design.

To request flight information, for example, consider the following possible prompts and slots they would fill:

Table 1: Sample prompts and natural language slots

Prompt	Slot
What city would you like to leave from?	origin
What city would you like to fly to?	dest
What date would you like to leave?	date
What time would you like to depart?	time
You're going from <*origin*> to <*dest*> on <*date*> at <*time*>. Is this correct?	confirm

If you have additional error or help prompts that can immediately precede recognition, you should write these as well, and take them into consideration when you write the grammars.

Note: The preceding prompts are appropriate for a directed dialog. A mixed-initiative dialog might instead start by asking "Where would you like to travel?" or "How can I help you?" and then pose more specific questions to obtain the missing pieces of information. It is much more difficult to predict the range of responses to an open-ended question. This makes the grammar more difficult to write and tune. However, mixed-initiative dialogs are closer to human interactions and can reduce the number of caller-system interactions. It is up to you to decide whether the dialog will elicit only simple responses or more complex ones.

Determine the appropriate type of grammar to use

When developing an application with the Nuance System, you can write your grammars using the following grammar formats:

- Nuance Grammar Specification Language (GSL)
- XML format of the W3C speech recognition grammar specification (GrXML)

- Statistical language model (SLM)

GSL is a Nuance format for defining the words and phrases that can be recognized, and optionally, the natural language commands associated with those phrases.

The World Wide Web Consortium (W3C) has proposed a syntax for speech recognition grammar formats: augmented BNF (ABNF) and XML. The Nuance System supports W3C grammars in the XML format (GrXML), which uses XML elements to represent the grammar constructs.

The GSL and GrXML formats provide two different syntaxes to represent the same underlying grammar. These formats are known as *probabilistic finite-state grammars*. GSL and GrXML grammars are useful when the application's prompts are sufficient to restrict the user's response. In general, GSL and GrXML grammars are appropriate for most applications.

Unlike a GSL or GrXML grammar, an SLM grammar is not manually written but trained from a set of examples that models the user's speech. To train an SLM grammar, you pass this set of examples (and optionally a domain-specific vocabulary) to a Nuance utility, which estimates the model probabilities.

SLM grammars are appropriate for recognizing free-style speech, especially when the out-of-grammar rate is high for GSL or GrXML grammars. Consider an application that requires an open-ended prompt such as "Please state the nature of your problem." Not only can the user's response be highly variable and hard to predict, but it can also contain restarts, filled pauses (*um* and *uh*), and ungrammatical sentences. An SLM grammar would be very appropriate for such an application. SLM grammars allow users to speak freely to an application and have their sentences interpreted by the NL engine without having to write complex grammar rules covering the entire sentence.

Note: While this section introduces SLM grammars, the development process described does not apply to SLM grammars, but to probabilistic finite-state grammars (GSL and GrXML). SLM grammars are described in detail in Chapter 5.

Anticipate the caller responses

After designing your prompts, you can guess more accurately how callers will respond. Remember that the two most typical responses in a directed dialog will contain just one of the following:

- The information item by itself
- The literal response to the question wording

You should also consider that people tend to hesitate at the start and sometimes say “please” at the end.

Do not try to cover the entire spoken English language. Reducing the size of a grammar improves the speed and accuracy of the system. Grammars should be specific and constrain the recognition to the domain at hand. While out-of-grammar speech can be a large source of errors, trying to cover too much will degrade accuracy. In tasks where out-of-grammar rate is a problem, you should consider SLMs.

Taking these points into account, here are some guesses as to how callers might respond to each of the prompts in Table 1 on page 5.

Table 2: Caller responses

What city would you like to leave from?	
San Francisco	[the city name by itself]
I’d like to leave from San Francisco	[a literal response]
Uh, San Francisco	[initial hesitation]
San Francisco, please	[final “please”]
I’m leaving from San Francisco	[some additional possibilities]
Departing from San Francisco	
What city would you like to fly to?	
New York	[the city name by itself]
I’m flying to New York	[a literal response]
I’d like to fly to New York	[another literal response]
Uh, New York	[initial hesitation]
New York, please	[final “please”]
Going to New York	[some additional possibilities]
My destination is New York	
What date would you like to leave?	
May second	[the date by itself]
I’d like to leave on May second	[a literal response]
I’m leaving on May second	[a second literal response]

Table 2: Caller responses

Leaving May second	[a third literal response]
Um, May second, please	[hesitation + final “please”]
What time would you like to depart?	
2 pm	[the time by itself]
I’d like to depart at 2 pm	[a literal response]
I’m departing at 2 pm	[a second literal response]
Departing 2 pm	[a third literal response]
2pm, please	[final “please”]
You’re going from <origin> to <dest> on <date> at <time>. Is this correct?	
Yes	[“yes” by itself]
No	[“no” by itself]
Yes, that’s correct	[a literal response]
Yes it is	[a second literal response]
No, that’s not correct	[a third literal response]
No, it’s not	[a fourth literal response]
Yeah	[casual alternative]

If you are developing an application where the prompts are not sufficient to restrict the user’s response—for example, your main prompt is “How can I help you today?”—you may want to use SLMs. Since SLMs need a large set of example responses to train the grammar, you may need to develop a data collection system or a pilot to gather training examples. Typical users call the data collection system and answer the prompts; you can then collect their responses, transcribe them to a file, and train the grammar using that file.

Identify the *core* of the grammar

A grammar typically consists of a *core* portion that contains the most important information-bearing words—like cities, dates, and times—and a *filler* portion that contains additional expressions such as “I’d like to...” or “please.”

The core portion is often highly reusable, so it makes sense to define a *subgrammar* or *grammar rule*—a smaller grammar used in building up hierarchies within larger grammars—describing just the core portion of a grammar. Information that pertains to a particular grammar can then be added in a higher-level more specific grammar.

In the flight information example, the core subgrammars should describe cities, dates, times, and confirmation.

Identify the *filler* portion of the grammar

The filler portion of a grammar depends largely on the prompt wording. If you have considered the caller responses, as described in “Anticipate the caller responses” on page 6, then you start by replacing the core portion of each utterance, in the list of anticipated phrases, with the name of a core grammar.

The portion of the original responses that remains after replacement is, very likely, the filler part of your grammar.

In the flight information example, you could use the grammar rules *CITY* and *DATE*, leading to the following types of transformed phrases:

- What city would you like to leave from?

  CITY

  I’d like to leave from *CITY*

  Uh, *CITY*

  CITY, please

  (I’m) leaving from *CITY*

  (I’m) departing from *CITY*

- What date would you like to leave?

  DATE

  I’d like to leave on *DATE*

  I’m leaving on *DATE*

  Leaving *DATE*

  Um, *DATE*, please

In the above phrases, the parentheses are used to indicate the enclosed phrase is optional.

Note that it is a good practice to put the fillers into their own subgrammars (see the section below for an example).

At this point—once the core and filler portions have been clearly identified—you have nearly written the grammar. All you need to do is write the final grammar definitions.

Write the code for your grammars

To write the code for your grammars, you can use GSL, GrXML, or SLMs, as appropriate. Since the two grammars in the flight information example (departure city and date) are fairly simple and the callers' responses can be easily restricted, GSL and GrXML are appropriate to write the grammars.

Assuming that you have the *CITY* and *DATE* subgrammars, the GSL code for the grammars would look like the following:

```
DEPARTURE_CITY [
   CITY
   (i'd like to leave from CITY)
   (uh CITY)
   (CITY please)
   (?i'm leaving from CITY)
   (?i'm departing from CITY)
]
DEPARTURE_DATE [
   DATE
   (i'd like to leave on DATE)
   (i'm leaving on DATE)
   (leaving DATE)
   (um, DATE please)
]
```

The above example grammars use standard GSL syntax, which is fully described in Chapter 2. Parentheses indicate concatenation—all the terms must be spoken—and question marks indicate the item is optional.

The example above assumes that *CITY* and *DATE* are subgrammars defined elsewhere.

You can also put the fillers into their own subgrammars; for example:

```
DEPARTURE_CITY (?CityPreFiller CITY ?please)
DEPARTURE_DATE (?DatePreFiller DATE ?please)
```

Where *CityPreFiller* and *DatePreFiller* are subgrammars defined elsewhere; for example:

```
CityPreFiller [
   (i'd like to leave from)
   (uh)
   (?i'm leaving from)
   (?i’m departing from)
]
```

Note that capturing fillers in subgrammars can increase the grammar coverage beyond the initial approach. For example, the phrase “uh, San Francisco please” would now be covered by the grammar.

Add natural language commands

The next step, adding natural language commands to the grammar, is done using GSL in this example. Note the following points in the code fragments below:

- *c* and *d* are *variables*
- The expressions *CITY:c* and *DATE:d* set the variables *c* and *d* with the values returned by the subgrammars *CITY* and *DATE*, respectively
- The expressions *$c* and *$d* are references to the values of the corresponding variables
- The expressions *{<origin $c>}* and *{<date $d>}* fill the slots *origin* and *date* with the values held in the variables *c* and *d*, respectively

```
DEPARTURE_CITY [
   CITY:c
   (i’d like to leave from CITY:c)
   (uh CITY:c)
   (CITY:c please)
   (?i’m leaving from CITY:c)
   (?i’m departing from CITY:c)
] {<origin $c>}
DEPARTURE_DATE [
   DATE:d
   (i’d like to leave on DATE:d)
   (i’m leaving on DATE:d)
   (leaving DATE:d)
   (um DATE:d please)
] {<date $d>}
```

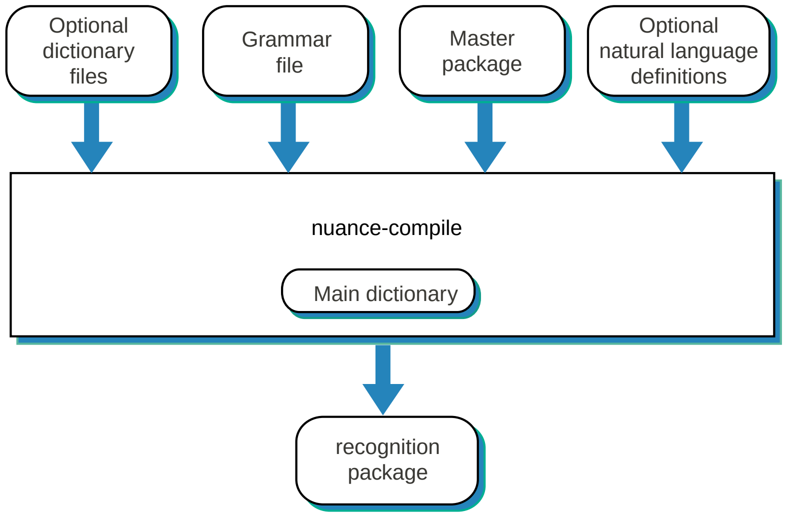
Building a recognition package

A *recognition package* contains the information to configure the Nuance Speech Recognition System for a specific application, including:

- Definitions of recognition grammars

- Pronunciations for the words in those grammars
- Pointers to the master packages selected for the application

It can also contain natural language processing information, to be used by the Nuance System's natural language processing engine (see Chapter 4). You generate recognition packages by creating the correct specification files and then compiling them with the command-line program *nuance-compile*. (The different compilation options are described in Chapter 6, "Compiling grammars.")



The two required inputs to *nuance-compile* are:

- A grammar file name
- A master package set name

The grammar file defines the recognition grammars used in an application. Each grammar describes a set of phrases that the Nuance System will consider during the recognition process. The structure and contents of a grammar are discussed in Chapter 2.

The master package defines, among other things, the languages that an application will use. Master packages differ in their accuracy, memory requirements, and computational requirements. Master packages are discussed in Chapter 6.

Optional inputs to *nuance-compile* such as natural language processing commands, a slot definitions file, and dictionary files containing phonetic pronunciations for words used in the grammars are also discussed in Chapter 6.

Chapter 2

Defining GSL grammars

2

The grammars that an application uses for recognition are defined in grammar files. Each grammar describes a set of word sequences that the Nuance System evaluates during the recognition process. A grammar can be as simple as “yes” versus “no,” as large as a list of all the names of people living in a city, or complex enough to support a dialog that registers a user in a course or traces a package for a delivery company.

This chapter describes how to write a GSL grammar.

Chapter 4 describes more complex grammar-writing techniques that include natural language interpretation.

Creating a grammar document

You write grammar definitions using a special language called the Grammar Specification Language, or GSL. You specify a GSL grammar in a *grammar document*. A grammar document is a text file—it can contain more than one grammar definition, and it has the file extension *.grammar*, for example, *myApp.grammar.* A grammar file is the basic component necessary to build a recognition package.

At the top level a grammar definition has the format:

GrammarRuleName GrammarDescription

where *GrammarRuleName* is the name of the grammar being defined and *GrammarDescription* defines the contents of the grammar. The simplest example of a grammar is one whose description is a single word:

```
.Account checking
```

The grammar file name has the format *package_name.grammar*, where *package_name* is the name of the recognition package you want to create. For example, if you create a grammar file called *banking.grammar*, the resulting recognition package is named *banking*.

The grammar *package_name.grammar* is the *primary* grammar file since it is the file passed to the compiler. The primary grammar file, however, need not be the *only* grammar file in your project, as GSL provides a directive that lets you refer to other grammar files within a grammar file.

Note: A formal definition of all the constructs available in GSL is included in Appendix A.

Grammar names

GrammarRuleName is the character string that other grammars or an application use to reference the named grammar. Grammar names must contain at least one uppercase character—typically the first alphabetic character—and can be up to 200 characters in length. All grammar names passed to the compiler must be distinct—that is, a grammar name can only have one grammar description associated with it.

The following characters are allowed in a grammar name:

- Uppercase and lowercase letters
- Digits
- The special characters:
    - - (dash)
    - _ (underscore)
    - ‘ (single quote)
    - @ (“at” sign)
    - . (period)

Other characters are not allowed.

Grammar descriptions

A *GrammarDescription* consists of a sequence of word names, grammar names, and operators that define a set of recognizable word sequences or phrases. Grammar and word names must be separated from one another by at least one white space character—space, tab, or newline.

Word names are the terminal symbols in a grammar description. Word names are lowercase character strings that correspond directly to the actual words spoken for recognition. For example, the word name *dog* corresponds directly to the spoken word "dog."

Word names cannot contain any uppercase letters, but can contain the other special characters that are allowed in grammar names (digits, "-", "_", and so on, as listed in "Grammar names" on page 14). You can include other special characters (except for white space and double quotes) if you enclose the word in double quotes. For example, "foo*bar" defines a legal word, but "foo bar" does not.

You can add comments in a grammar description by using a semicolon (;)—all text in a line after a semicolon is ignored by the compiler. Comments can be included anywhere in a grammar file (or in any of the other package files mentioned in this manual).

You construct a grammar description by using a set of five basic grammar operators: ( ), [ ], ?, +, and *, described in Table 3. White space is optional between operators and operands (grammar or word names).

The symbols A, B, C, and D in the following table denote a grammar or word name.

Table 3: Grammar operators

Operator	Expression	Meaning
( ) *concatenation*	(A B C ... D)	A and B and C and ... D (in that order)
[ ] *disjunction*	[A B C ... D]	One of A or B or C or ... D
? *optional*	?A	A is optional
+ *positive closure*	+A	One or more repetitions of A
* *kleene closure*	*A	Zero or more repetitions of A

Here are some simple GSL expressions and some of the phrases they describe:

```
[morning afternoon evening]
```

"morning", "afternoon", "evening"

```
(good [morning afternoon evening])
```

"good morning", "good afternoon", "good evening"

```
(?good [morning afternoon evening])
```

“good morning”, “good afternoon”, “good evening”, “morning”, “afternoon”, “evening”

```
(thanks +very much)
```

“thanks very much”, “thanks very very much”, and so on

```
(thanks *very much)
```

“thanks much”, “thanks very much”, “thanks very very much”, and so on

Caution: The following strings are reserved for internal use and cannot be used to denote words or grammar names (*n* designates any integer):

`AND-`*n*, `OR-`*n*, `OP-`*n*, `KC-`*n*, `PC-`*n*

Grammar hierarchies

You can build a hierarchy of grammars using subgrammars, also called grammar rules. By breaking a grammar into smaller units, you can create components that are reusable by multiple grammars or applications.

The use of subgrammars:

- Simplifies grammar creation and revision
- Helps focus the grammar development to the task at hand
- Hides unnecessary details and promotes modularity

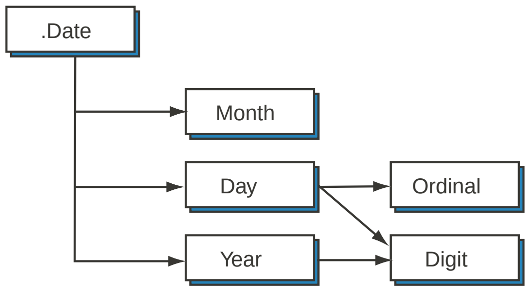
Top-level grammars and subgrammars

A grammar is either a *top-level* grammar or a subgrammar. A top-level grammar is, by definition, one whose name has a period (.) as its first character, for example, *.Account*.

Only top-level grammars can be referenced by an application at runtime. Any other grammar—one whose name does not begin with a period—is a subgrammar that can only be referenced by other grammars. A grammar named *.SENTENCE*, for example, can be referenced by an application or by other grammars, while a grammar named *SENTENCE* can only be referenced by other grammars.

Caution: The distinction between top grammars and subgrammars does not apply to grammars used dynamically, including just-in-time, VoiceXML, and SpeechObjects grammars.

Subgrammars let you define complex grammars as a hierarchy of smaller grammars. This simplifies your specifications and provides a more efficient way to specify grammars with shared internal structure. For example, a grammar defining how to say a date might have subgrammars for day, month, and year, and those subgrammars might reference subgrammars for different ways of saying numbers, and so on:



The top-level grammar *.Date* can then be referenced by applications for recognition, without needing to know anything about the subgrammars used by the *.Date* grammar.

To include the contents of a subgrammar in a grammar description, you just refer to the subgrammar by its unique name as you do with a word name. The description of the subgrammar referred to is included in exactly the location you specify it in.

For example, the following grammars describe *naturally phrased* numbers from 0 to 99. Four subgrammars are defined and then used to create the top-level grammar *.N0-99*, that can be referenced by applications.

```
; Sample GSL code for natural numbers from 0 to 99
; subgrammar for one of the non-zero digits
NZDIGIT [one two three four five six seven eight nine]

; subgrammar for one of the digits including zero
DIGIT [one two three four five six seven eight nine zero   oh]

; subgrammar for one of the teen words and ten
TEEN [ten eleven twelve thirteen fourteen fifteen sixteen
      seventeen eighteen nineteen]

; subgrammar for one of the decade words except ten
DECADE [twenty thirty forty fifty sixty seventy eighty
       ninety]
```

```
; the top-level grammar, which references the subgrammars
; defined above
.N0-99 [(?NZDIGIT DIGIT) TEEN (DECADE ?NZDIGIT)]
```

Phrases defined by the top-level grammar *.N0-99* include: “two,” “eight two,” “zero,” “seventeen,” “fifty,” and “thirty nine.”

Including grammar files

In many cases the grammars for a particular application are defined in a single main grammar specification file. However, you can use the `#include` directive to include a grammar file in a static grammar. This lets you create modular subgrammars that you can later include in the grammar file for one or more applications, keeping your application grammar files smaller and simpler. Included grammar files can contain both subgrammars and top-level grammars.

When the grammar file is compiled, any GSL line of the form

```
#include "filename.grammar"
```

is replaced by the contents of the file *filename.grammar*. The grammar compiler searches for your include files locally—that is, in the directory where the compiler is run. If you want to include a Nuance sample grammar, copy this grammar from the *grammars* directory (for example, *%NUANCE%\data\lang\*`language`*\grammars*) to the directory where the compiler is run.

An included file may itself contain `#include` lines. Using the `#include` directive is necessary when your grammars are defined in more than one file, since you can pass one file only as an argument to the command-line compiler.

Default grammar location

The default directory location for North American English grammars is *%NUANCE%\data\lang\English.America\grammars*.

Grammars for languages other languages are installed in the directories *%NUANCE%\data\lang\*`language`*\grammars*, where `language` indicates one of the supported natural languages—for example, *%NUANCE%\data\lang\Spanish.America\grammars*.

The grammar files installed in the directories *%NUANCE%\data\lang\*`language`*\grammars* cover common application vocabularies such as numbers, dates, money amounts, and confirmation (yes/no) responses. You can copy these files into your package directory and include them in your grammars, using them verbatim or editing them as needed.

The name format of the grammar files in the *English.America* directory is `<name>`*.grammar-v*`<vnumber>`, where `vnumber` denotes the version of the grammar. For example, to use the latest version the *money. grammar*, make a copy of that file into your working area, and then use the following directive in other files, as needed:

```
#include "money.grammar"
```

Caution: Be careful with the use of the include directive. Multiple inclusions of a grammar file result in compilation errors, as all grammar names passed to the compiler must have exactly one definition.

Recursive grammars

Recursive grammars—grammars that reference themselves—are only allowed in certain cases. Here is a simple example of a recursive grammar:

```
SENT [(see spot run) (SENT and SENT)]
```

Recursion can also occur indirectly, as in the following grammar description:

```
NounPhrase (NounPhrase1 *PrepositionalPhrase)
NounPhrase1 (Determiner Noun)
Determiner [the a]
Noun [cat dog]
Preposition [from with]
PrepositionalPhrase (Preposition NounPhrase)
```

The NounPhrase grammar is recursive because it references *PrepositionalPhrase*, whose description references back to *NounPhrase*.

GSL does not support *left-recursive* grammars. Left recursion occurs when the self reference (direct or indirect) is located in the leftmost position. Any other type of recursion is valid in GSL.

For example, the following grammar will not compile because it is left recursive:

```
Digits (Digits Digit)
```

But the following one is right recursive, and will therefore compile:

```
Digits (Digit Digits)
```

This next one is middle recursive, and will also compile:

```
Digits (Digit Digits end)
```

Indirect left recursion is also prohibited, for example:

```
NounPhrase (Determiner Noun)
Determiner [ the
              a
              NounPhrase ]
```

The previous grammar is prohibited because *Determiner* appears in the initial position within *NounPhrase*, and *NounPhrase* appears in the initial position within *Determiner*. Note that while *NounPhrase* does not literally appear in the initial (leftmost) position in the definition of *Determiner*, it is treated as such because it appears in an `OR` construction where the order is not meaningful. An equivalent definition for the *Determiner* subgrammar could also be:

```
Determiner [ NounPhrase
               a
             the ]
```

where the *NounPhrase* subgrammar more obviously appears in the leftmost position.

There is an important requirement to remember when you are compiling a valid recursive grammar (that is a non-left recursive grammar): you *must* use the *-dont_flatten* compiler option. If *-dont_flatten* is omitted from the compilation options, you get a fatal error (stating that a grammar is recursively calling itself) even when you are using valid recursion in that grammar. See "Creating unflattened grammars" on page 112 for details on this compiler option.

Grammar weights and probabilities

The recognition engine can use weights and probabilities while searching for matches in the space of allowable utterances.

Using weights

A *weight* is a multiplying factor that influences the likelihood of a phrase. Weights cause the recognition server to favor certain phrases over others. Typically you assign higher weights to phrases expected to be spoken more frequently.

Using grammar weights enables the recognition system to favor certain results over others when the acoustic processing results in similar scores. For example, the time "one fifty" may be acoustically confusable with "one fifteen," but may be spoken less frequently. In the case where 1:50 and 1:15 have a similar score, using weights should reduce the recognition error rate to favor 1:15.

You assign weights to GSL constructs by using the tilde character (~). The GSL syntax to specify a weight for an item C is:

`C~`*`weight`*

where `weight` is a non-negative number. Weights can be used in disjunct (OR) constructs. For example, the following *City* grammar assigns different weights to each of the four city names:

```
City [
   boston~1.4
   (new york)~1.2
   dallas
   topeka~.4
]
```

Not specifying a weight is the same as specifying a weight of 1.0 for the item. In the example above, the weight used for "`dallas`" is implicitly set to 1.0 and is therefore equivalent to "`dallas~1.0`". A weight value below 1.0 decreases the likelihood of the phrase with respect to phrases without weights. Similarly, a value above 1.0 increases the likelihood.

Using weights with special operators

Weights can also be assigned to the operands of optional (?), kleene closure (*), and positive closure (+) operators. These values indicate the probability of following or not following the closure. For example:

Expression	Meaning
? A~.6	A is 60% likely
+ A~.6	The probability of any additional occurrence of A (after the first one) is 60%
* A~.6	The probability of each occurrence of A (including the first one) is 60%

Using probabilities

The Nuance System gives you the flexibility to set weights in disjunct (OR) constructs so that they represent probabilities. You do this by constraining the weights assigned to the set of paths in the OR construct so that the sum of the weights equals one.

As with weights, you assign probabilities by using the tilde character (~). The GSL syntax to specify a probability for a construct C is:

```
C~prob
```

where `prob` is a number between 0 and 1. For example:

```
City [
   boston~0.4
   (new york)~0.3
   dallas~.2
   topeka~.1
]
```

Probabilities can be very useful to reflect the frequency of items in a construct.

Guidelines for using weights and probabilities

When assigning overall and individual weights and probabilities, note that the following two specifications are equivalent:

```
[A~.3 B~.3 C~.1]~.5
[A~.15 B~.15 C~.05]
```

Nuance recommends that, if you choose to use grammar weights and probabilities, you test your application's performance both with and without them. The *nuance-compile-ngo* program has the option *-dont_use_grammar_probs*, which lets you compile a package ignoring any weight and probability specifications found in your grammars. This feature is useful when you want to compare recognition performance of a package with and without the use of weights and probabilities.

Note: Nuance provides *compute-grammar-prob*, a command-line tool that assign probabilities to GSL grammars based on data that you provide. See the *Nuance System Administrator's Guide* for more information.

Adding weights and probabilities to your grammar can potentially increase both recognition accuracy and speed—however, assigning bad values can actually hurt recognition performance. Weights can be especially effective for improving the recognition performance on lists of items (for example, cities, equities, and so on). In these cases, it is important to estimate the weights based on sufficient, real-life usage data. A rule of thumb would be to specify the weights based on at least ten samples (on average) for each list element.

Important note

When using weights and probabilities in complex grammars, it is important to note the implicit weight of 1.0 for the non-labelled paths in the grammar. For example, a grammar may have two rules in parallel, where one rule uses weights and the other does not. A possible unintended consequence is that the rule not using weights may have stronger likelihood than the rule using weights. Try to be consistent in your usage of weights and probabilities throughout all the grammars in your application.

Identifying a text string as Nuance GSL

To specify that a text string is actually a set of grammar rules in Nuance GSL, you can use the `;GSL2.0` header, as follows:

```
;GSL2.0
TEST_SENTENCE:public (the quick COLOR fox jumped over the TYPE
dog)
COLOR [red blue brown]
TYPE [lazy diligent]
```

This header specifies that the content that follows is a set of Nuance GSL rules. For example, the sample code above identifies `TEST_SENTENCE`, `COLOR`, and `TYPE` as GSL rules.

The header can be specified inside a Nuance GSL file, as shown above, or to an API function directly when using a just-in-time grammar, as shown in the following example:

```
Recognize(";GSL2.0
          PEOPLE:public <http://server/people.gsl>");
```

Nuance strongly recommends that you begin to use it in all your GSL grammars; it will become required in a future version.

Language-specific grammar conventions

Through deployment experience, Nuance has developed some general recommendations and language-specific conventions that you should conform to when writing grammars. For the most up-to-date information for the language you are working with, see the Nuance Technical Support Online website at *support.nuance.com*.

Chapter 3

Dynamic grammars: Just-in-time grammars and external rule references

3

A *dynamic grammar* is a grammar that can be dynamically created and modified while an application is running. The Nuance dynamic grammar mechanism lets you create and update grammars at runtime and use them for recognition immediately, without needing to recompile the recognition package and start a new *recserver* process.

A dynamic grammar can be a file referenced using external rule references, or it can be created directly in a database using API functions. This chapter first introduces dynamic grammars created using API functions. It then describes how to refer to dynamic grammars using external rule references and how to use just-in-time grammars. It also provides information on configuring your system to use external rule references and just-in-time grammars.

Creating a dynamic grammar in a database

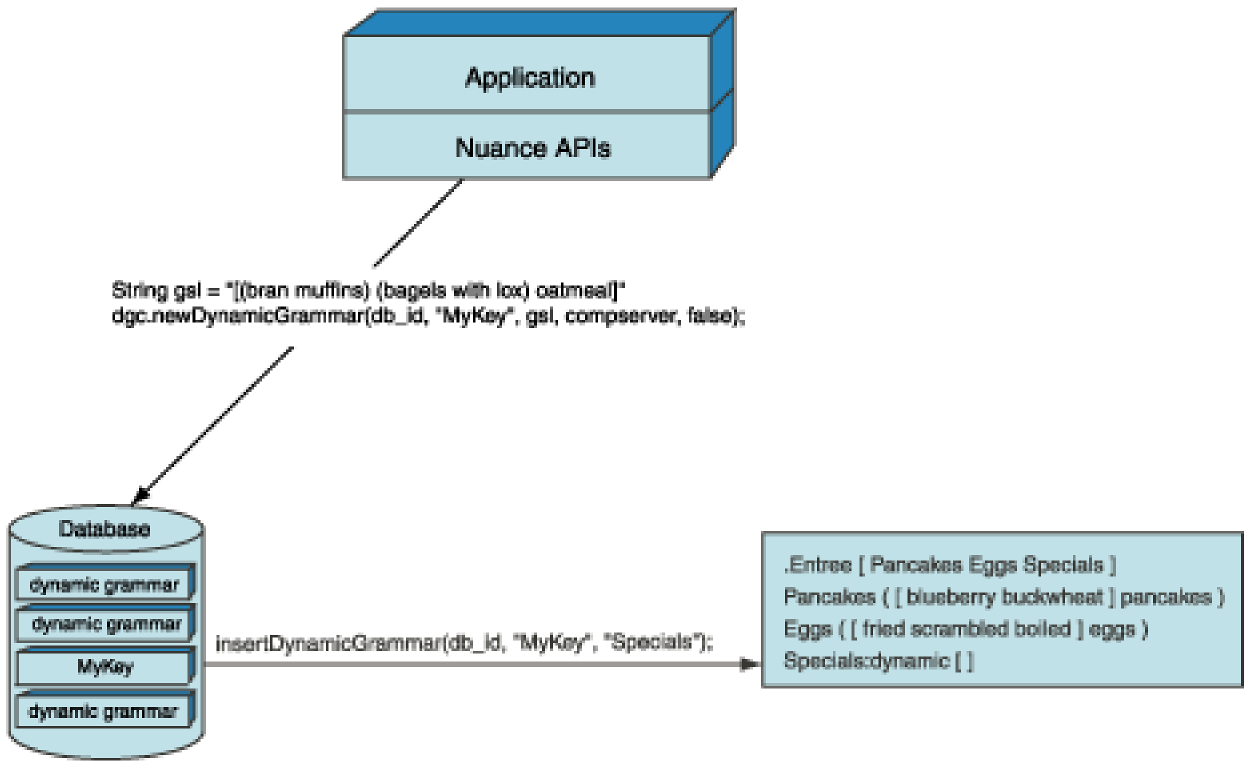
You can create a GSL or W3C XML (GrXML) grammar directly in a database using API functions. To create a dynamic grammar directly in a database, you first add the grammar as a component in a top-level grammar using the following GSL syntax:

```
GrammarName:dynamic []
```

This specification acts as a placeholder for the dynamic grammar that your application inserts at runtime. For example, this shows a simple GSL file that specifies a grammar for a breakfast menu that includes daily specials:

```
.Entree [ Pancakes Eggs Specials ]
Pancakes ( [ blueberry buckwheat ] pancakes )
Eggs ( [ fried scrambled boiled ] eggs )
Specials:dynamic []
```

When you compile the grammar *.Entree*, it contains the phrases “blueberry pancakes,” “buckwheat pancakes,” “fried eggs,” “scrambled eggs,” and “boiled eggs,” but the *Specials* subgrammar is empty. To fill the *Specials* grammar at runtime, your application finds (or creates, as shown in the example below) the correct dynamic grammar in the appropriate database and inserts it into the *.Entree* grammar at the location *Specials*:



```
String gsl = "[(bran muffins) (bagels with lox) oatmeal]"
dgc.newDynamicGrammar(db_id, "MyKey", gsl, compserver, false);
```

```
insertDynamicGrammar(db_id, "MyKey", "Specials");
```

```
.Entree [ Pancakes Eggs Specials ]
Pancakes ( [ blueberry buckwheat ] pancakes )
Eggs ( [ fried scrambled boiled ] eggs )
Specials:dynamic [ ]
```

After inserting this dynamic grammar, the grammar *.Entree* contains the phrases “bran muffins,” “bagels with lox,” and “oatmeal” in addition to its previous contents. You can specify whether these phrases should remain in the grammar indefinitely or only for the duration of the current call.

A dynamic grammar can also fill *slots*. For information about slots, see Chapter 4.

Note: You can have up to 255 `:dynamic` labels in a package. Also, a dynamic grammar must be a subgrammar in another top-level grammar—it cannot be a top-level grammar itself. It can, however, be as simple as:

```
.DynamicGrammar MyDynamicGrammar
MyDynamicGrammar:dynamic []
```

In this case you can perform recognition using the top-level grammar *.DynamicGrammar*, whose contents are completely dynamic.

You must compile a dynamic grammar and a static grammar using the same master package. For complete details on creating GSL and GrXML dynamic grammars, see the *Nuance Application Developer's Guide*.

You cannot have a dynamic grammar in one package and a top-level grammar—of which that dynamic grammar is a subgrammar—in another package, and use both grammars at once. Both dynamic and static grammars must be compiled into a single package, if they are to be used together.

Note: With Nuance 8.5, information kept in a dynamic grammar database can instead be kept in a grammar on file and loaded on demand, as provided by the just-in-time and external rule references grammar features. If your application requires the reliability, redundancy, and scaling associated with a commercial database, Nuance recommends that you keep your grammars in a dynamic grammar database. Otherwise, a just-in-time grammar or a dynamic grammar referenced using an external rule reference may be a more flexible approach.

External rule references and redefinitions

External rule references let you store dynamic grammars not only in a database but also in a file system or on a web server.

This section first describes how to reference and redefine external rules. It then presents the external grammar reference syntax.

Note: Your system configuration must include a web server for the external rule references to work. See "Configuring your system for external rule references and just-in-time grammars" on page 38 for more information.

Referring to external grammar rules

From a just-in-time GSL grammar, you can refer to grammar rules in another grammar file. Nuance provides the following access modes to refer to external grammar rules:

- `http:`—points to a grammar on a web server
- `file:`—points to a grammar on disk
- `dgdb:`—points to a grammar in a dynamic grammar database
- `static:`—points to a precompiled static grammar
- `local:`—points to a public rule in the same grammar document

- `special:`—points to a grammar that modifies the search probabilities
- `fail:`—specifies a failure condition

The access modes are described below.

Using http: or file: URIs

To refer to a grammar residing on a web server, specify the grammar URI inside angle brackets, as shown in the following example:

```
;GSL2.0
TEST_SENTENCE:public (the quick COLOR fox jumped over the
   <http://types.com/todayslist.gsl> dog)
COLOR [red blue brown]
```

This GSL code tells the recognition server to get the list of types, at recognition time, from the specified `http:` URI and insert it in the specified grammar. Each time the *TEST_SENTENCE* grammar is used, the latest version of *todayslist.gsl* is fetched.

To refer to an external grammar rule on disk, specify the keyword `file:` followed by the location of the grammar, and place this reference inside angle brackets, as shown in the following example:

```
;GSL2.0
TEST_SENTENCE:public (the quick COLOR fox jumped over the
   <file:/c:/user/local/types.gsl> dog)
COLOR [red blue brown]
```

This GSL code gets the list of types from the specified file location and inserts it in the current grammar.

The `http:` and `file:` URIs can refer to the following types of files:

- GSL files, with the *.gsl* extension

  ```
  http://gramserv.com/groceries.gsl
  file:/c:/tmp/groceries.gsl
  ```

- W3C XML (GrXML) grammar files, with the *.grxml* extension

  ```
  http://gramserv.com/groceries.grxml
  file:/c:/tmp/groceries.grxml
  ```

- Nuance Grammar Object (NGO) files, with the *.ngo* extension

  ```
  http://gramserv.com/groceries.ngo
  file:/c:/tmp/groceries.ngo
  ```

Nuance provides the Nuance Grammar Object (NGO) grammar format. A Nuance Grammar Object is a grammar that is precompiled using the *nuance-compile-ngo* utility. This utility can compile a GSL, GrXML (W3C format), or SLM grammar file into a binary grammar file with the *.ngo* extension. The compiled file can then be loaded on demand by a just-in-time grammar. Since an NGO file is precompiled, it is faster to load at runtime.

See "Dynamic compilation to a file" on page 98 for more information about the *nuance-compile-ngo* utility.

Using dgdb: URIs

You can refer directly to compiled dynamic grammars in a dynamic grammar database by specifying the keyword `dgdb:` followed by the database key and database descriptor. The syntax for the `dgdb:` URI is:

```
dgdb:?key=val1&dbdesc=val2
```

where *`val1`* is the database key and *`val2`* is the database descriptor for the grammar. Both *`val1`* and *`val2`* must be URI-encoded, which means that no illegal characters are used. The list of characters that cannot be used include all characters other than:

- a-z
- A-Z
- 0-9
- ; / ? : @ & = + $, - _ . ! ~ * ' ( )

The illegal characters must be converted to hexadecimal and written as %xx. For example, the space character, ASCII 32, becomes "%20". For more information about URI encoding, see RFC 2396 (*www.ietf.org/rfc/rfc2396.txt*, section 2.4)

For example, in the following reference, the portion in bold must be URI-encoded:

dgdb:?key=PeterGrammar&dbdesc=**provider=fs,root=/usr/dbs,name=db1, class=dgdb**

To convert a DBDescriptor object to the URI-encoded string required by the `dgdb:` URI scheme, use the Nuance DBDescriptor function `DBDescriptorToQueryString()`. See the *Nuance API Reference* for more information about this function.

Using static: URIs

A grammar can explicitly refer to a `:dynaref` or `:public` rule in the host package by using the `static:` keyword. For example, in the following code, the grammar refers to the static grammar rule *OtherInterestingSentences*:

file:/c:/usr/grammars/other.gsl

```
;GSL2.0
TEST_SENTENCE:public
[(the quick brown fox)
 <static:OtherInterestingSentences>
]
```

The static grammar rule to which the grammar refers must be identified by the `:dynaref` or the `:public` keyword in the static grammar file. For example, the static grammar rule *OtherInterestingSentences* in the example above would have been defined as following in a grammar file, and then compiled with *nuance-compile*:

```
OtherInterestingSentences:dynaref [...]
```

To compile these grammars into your application's recognition package, you must use the *nuance-compile* option *-dont_flatten*.

Note that, when compiling the grammar that refers to a static grammar rule, the compiler does not check that the static grammar exists. For example, when compiling the above grammar *other.gsl*, the compiler does not check that grammar *OtherInterestingSentences* exists. This grammar must be present only when it is used at recognition time.

Fetching grammars

An external reference is not evaluated until recognition time. If the contents of a grammar change over time, the latest version of the grammar is fetched for each recognition. The Nuance implementation of grammar fetching is very efficient and is designed to minimize network traffic. Grammar fetching is configured with the parameter `config.EGRCacheMinFreshSecs`. This parameter specifies the time (in seconds) that a grammar referred with an `http:` or `file:` URI is assumed to be *fresh*. The default value is 10 seconds.

This parameter works as follows. When an application refers to the grammar, the compilation server loads it, compiles it, and stores the compiled version in its cache. For the next 10 seconds (or as specified by `config.EGRCacheMinFreshSecs`), all other requests for that grammar are served out of the cache without testing whether the source file has changed. The first request for that grammar after the 10 seconds have expired forces the compilation server to check the eTag of the grammar, via an HTTP request, or the timestamp of the grammar file. If the grammar has not changed, the cached copy is used for another 10 seconds. If the grammar has changed, it is fetched and recompiled.

This feature provides a nice way to minimize network usage. No matter how many times a grammar is requested (across any number of ports simultaneously), the grammar is examined only once every 10 seconds.

The `config.EGRTimeoutMs` and `comp.ExternalReferenceTimeoutMs` parameters are also used to configure grammar fetching. The `comp.ExternalReferenceTimeoutMs` specifies the maximum number of milliseconds allowed to fetch an individual grammar document. The `config.EGRTimeoutMs` parameter specifies the maximum number of milliseconds that are allotted for fetching all the external grammar components for a just-in-time recognition. Any requests that are still outstanding once the timeout is reached are reported as load failures with a “Timed out waiting for HTTP response” annotation in the recognition server log. These references are replaced with the backoff reference. See “Specifying a grammar backoff” on page 34 for more information.

Referring to a specific external rule

A grammar file often contains multiple grammar rules. When referring to an external grammar file in a grammar, if you do not specify a rule, the URI refers to the *root* rule by default. For GSL files (and NGO files compiled from GSL files), the root rule is defined as the first `:public` rule in the file.

Note: For W3C grammars (and NGO files compiled from W3C grammars), the root rule is determined according to the W3C grammar specification, described in *Speech Recognition Grammar Specification for the W3C Speech Interface Framework*. *www.w3.org/TR/2001/WD-speech-grammar-20010820/* for more information. Nuance is currently working from the draft dated **August 20, 2001**.

You can also refer to a specific rule in a grammar file by adding the # symbol followed by the rule name, as shown below:

`http:` URI	`http://URI_of_grammar#RuleName`
	For example
	`http://gramserv.com/groceries.gsl#Nuts`

`file:` URI	`file:/location_of_grammar#RuleName`
	For example
	`file:/c:/usr/grammars/groceries.gsl#Nuts`

`dgdb:` URI	`dgdb:DBDescriptor_and_key#RuleName` For example dgdb:?key=PeterGrammar&dbdesc=provider=fs,root=/usr/dbs,name=db1,class=dgdb#`Nuts`

Defining a rule as public

When making a reference to a specific external rule, you can specify a public rule only. A rule is defined as a *public* rule if it is followed by the `:public` or `:dynaref` keyword. For example, all of the rules in this GSL file are public:

```
;GSL2.0
MyGrammar1:public [...]
MyGrammar2:dynaref [...]
```

In a GSL file, if none of the rules in a grammar file are declared public, then all rules are automatically made public, and the first rule becomes the root rule. In this case, a warning message is displayed. For consistency and clarity, Nuance recommends that you explicitly identify the rules that you want to make public with the `:public` keyword.

Examples

For example, in the following grammar files, both *Fruits* and *Nuts* are public rules, and the *Fruits* rule is the root rule.

http://server/groceries.gsl

```
;GSL2.0
Fruits [apple orange]
Nuts [peanut cashew]
```

file:/c:/usr/grammars/groceries.gsl

```
;GSL2.0
Fruits:public [apple orange]
Nuts:public [peanut cashew]
```

Given the files above, the following URI refers to the *Fruits* rule:

```
http://server/groceries.gsl
```

While the following URI refers to the *Nuts* rule:

```
file:/c:/usr/grammars/groceries.gsl#Nuts
```

Finally, consider the following example:

http://server/groceries.gsl

```
;GSL2.0
Fruits [apple orange]
Nuts:public [peanut cashew]
```

In this example, only the *Nuts* rule is public. Therefore, you cannot refer to the *Fruits* rule directly from another grammar. Given this file, the following URI refers to the *Nuts* rule:

```
http://server/groceries.gsl
```

Redefining an external rule

You can dynamically redefine an external dynamic rule in a grammar by using the `!` symbol, as follows:

```
< <Grammar_File_URI> ! RuleName = <New_URI> & ... >
```

where `RuleName` is tagged with the `:dynamic []` keyword in the grammar file specified by `Grammar_File_URI`. Redefining an external rule is very useful in personalized applications, where the grammars need to be customized per users.

For example, consider the following grammar file for a personal dialing application:

file:/c:/usr/local/template.gsl

```
;GSL2.0
MAIN_MENU:public [ (please call MyFriends)
             help
             cancel
             Other_Cmds ]
Other_Cmds:dynamic []
MyFriends:dynamic []
```

A grammar can refer to this file and redefine the *Other_Cmds* and *MyFriends* rules dynamically as follows:

```
;GSL2.0
My_Main_Menu:public < <file:/c:/usr/local/template.gsl> !
      Other_Cmds = <file:/c:/usr/local/cmd.gsl> &
      MyFriends = <http://listserver/user18257_contactlist.ngo>>
```

In this example, the *Other_Cmds* rule is redefined with a grammar available on a file system, while the *MyFriends* rule is redefined with the grammar for the caller's contact's list. Note that you can also redefine an external rule to one that is defined locally using the `local:` keyword. See "Redefining an external rule to a local rule" on page 46 for more information.

An external rule redefinition is very similar to a dynamic grammar insertion, but it has many advantages:

- The inserted grammar can come from any type of URI, not only from a dynamic grammar in a database
- It is specified in the grammar itself; using a dynamic grammar function call is not required
- It is not persistent—it lasts for one utterance only

Note that any grammar rule previously inserted with the `InsertDynamicGrammar()` function is cleared after an external rule redefinition. Nuance does not recommend that you mix external rule redefinitions with dynamic grammar insertion functions in your application.

Specifying a grammar backoff

You can specify a *grammar backoff*—the alternate grammar to use if a grammar referenced using a `file:`, `http:`, or `dgdb:` URI cannot be found.

To specify the grammar backoff, you use the `%` symbol, as follows:

```
< <URI_1> % <backoff_URI> >
```

For example:

```
;GSL2.0
MAIN_MENU:public (<<http://polite.com/nice.gsl > %
                <file:/c:/user/local/MyNice.gsl>>
                get me a quote for STOCK)
STOCK:dynamic []
```

In this example, the *nice.gsl* grammar will be replaced by the local grammar file `MyNice.gsl` if the HTTP request fails. If the local file is not found, it is substituted with the global grammar backoff defined with parameter `egr.Backoff`. This parameter specifies a default backoff grammar that is used when any URI reference fails, even if you do not use the `%` symbol. To do so, you set parameter `egr.Backoff` to the backoff grammar.

You can also use the following special references as grammar backoffs:

- `special:passthrough`—defines a rule that is automatically matched, without the user speaking any word
- `special:roadblock`—defines a rule that can never be spoken
- `special:resistor`—lets you change the probability that any given rule will be taken instead of simply disabling the rule

The following example shows how special references can be used as grammar backoffs.

```
;GSL2.0
MAIN_MENU:public
      (?<<http://polite.com/nice.gsl> % <special:passthrough>>
         get me a quote for
         <<http://broker.com/stock.gsl> % <special:roadblock>>)
```

In this example, if the first `http:` request fails, the *nice.gsl* grammar is replaced by a passthrough and is therefore simply ignored. However, if the *stock.gsl* grammar cannot be found, it is replaced by a roadblock, which means that recognition will never happen for the *MAIN_MENU* rule.

For more information about passthroughs and roadblocks and about the `special:resistor` keyword, see "Using the dynamic grammar gate technique" on page 49.

Failed references

To cause a recognition to fail immediately and return an EXCEPTION RecResult, you can use the `fail:` reference. The text of the recognition result will include all the grammar references that generated a failure as well as their source location.

For example, consider the following *http://up/foo.gsl* grammar:

```
;GSL2.0
Foo:public <http://down/foo.gsl>
```

Then consider the following just-in-time grammar:

```
;GSL2.0
ROOT:public (<http://down/foo.gsl>
             <<http://down/bar.gsl> % <fail:died>>
             <http://up/foo.gsl>)
```

Suppose that the `http://down` server is down. Since the just-in-time grammar includes a `fail:` reference, trying to perform recognition with the just-in-time grammar will create an EXCEPTION RecResult similar to the following:

```
<http://down/foo.gsl> failed: couldn't load
```

```
<<http://down/foo.gsl> % <fail:died>> failed
In <http://up/foo.gsl>, <http://down/foo.gsl> failed: couldn't
load
```

The first reference failed because the `http://down` server is down. The second reference also failed, but because we used the `fail:` keyword, the `<fail:..>` reference is also included in the RecResult, providing more information as to why the reference failed. The third reference failed because the *http://up/foo.gsl* grammar contains a reference to a grammar on the server that is down.

Note that if you do not specify a grammar backoff as described in "Specifying a grammar backoff" on page 34, the default `fail:` reference "`<fail:couldn't load>`" is used, as specified by parameter `egr.Backoff`.

Relative paths in external rule references and redefinitions

When using the `file:` and `http:` URIs, a *relative path* is relative to the grammar file that contains the reference.

For example, consider the file *file:/c:/tmp/head.gsl* which contains the following string:

file:/c:/tmp/head.gsl

```
...<tail.gsl>...
```

An external rule reference to the file *file:/c:/tmp/head.gsl* will try to load the file *file:/c:/tmp/tail.gsl*.

Similarly, consider the file *http://megacorp.com/dave/test.gsl* which contains the following string:

http://megacorp.com/dave/test.gsl

```
...<../top.gsl>...
```

An external rule reference to *http://megacorp.com/dave/test.gsl* will try to load the file *http://megacorp.com/top.gsl*.

Note that for grammar backoffs and external rule redefinitions, a relative path is relative to the grammar file containing the backoffs and external rule redefinitions, not to the rules they are referencing of redefining.

For example, consider the file *file:/c:/tmp/dave/top.gsl* which contains the following string:

```
...<<<file:/c:/nuance/car.gsl> ! Mine=<hiscars.gsl>>
%<allcars.gsl>>...
```

An external rule reference to *C:\tmp\dave\top.gsl* will resolve references to files *hiscars.gsl* and *allcars.gsl* in the *C:\tmp\dave* directory, and not in the *C:\nuance* directory.

If there is no parent insertion, a relative path is resolved as follows:

- If the relative reference has an `http:` scheme, it is not resolved. Nuance does not recommend using a relative `http:` reference.
- If the reference has a `file:` scheme, and parameter `egr.file.context` is not set, the relative path is relative to the directory in which the compilation server was started. If parameter `egr.file.context` is set, all file references (whether relative or not) are resolved relative to that path. See "Parameters for external rule references" on page 39.
- If the reference does not have a scheme, then it is resolved only when a `Content-Base` key is specified in the `RECOGNIZE` header of the JIT grammar, as described in "Specifying just-in-time grammar options: Using the keyword RECOGNIZE" on page 42. For example, consider the following grammar:

  ```
  RECOGNIZE
  Content-Base: http://up/here/

  ;GSL
  ROOT: <foo.gsl>
  ```

  In this example, the `<foo.gsl>` reference is resolved as `<http://up/here/foo.gsl>`.

Specifying builtin: grammars

You can create built-in grammars and reference them using the `builtin:` URI scheme. To use this scheme, you set the `egr.builtin.context` parameter to the location of the built-in grammars, for example "`http://tango:8080/`". You then use the `builtin:` URI as follows:

```
Rule:public (i want <builtin:grammar/digits> scoops of ice cream)
```

To find this grammar, the recognition server takes the string that follows the `builtin:` keyword and prepends the value of the parameter `egr.builtin.context`. For example, if parameter `egr.builtin.context` is set to "`http://host/`", the expression `<builtin:grammar/digits>` is expanded to `<http://host/grammar/digits>.`

Configuring your system for external rule references and just-in-time grammars

This section provides information on configuring your system for external rule references and just-in-time grammars.

Setting up a web server for external rule references

For the URI-based external rule references to work, your system configuration *must* include a web server.

You need to define the MIME content types for the grammar files as follows. Refer to your web server documentation for information on setting MIME types.

Grammar file extension	MIME type
.gsl	`application/x-nuance-gsl`
.ngo	`application/x-nuance-dynagram-binary`
.grxml	`application/grammar+xml`

The Voyager handles downloaded grammar documents as follows. It first looks at the web server content type. If it's one of the content types defined above, it handles the file appropriately. If it's an unknown content type, it looks at the file extension to determine how to handle the file. If that fails, it examines the document header: a header of `;GSL2.0` indicates a GSL file, while a header of `<?xml ...>` indicates a W3C XML file. If that fails, the Voyager resolves the reference in a platform-specific manner. If the grammar document type cannot be resolved, a failure is reported.

Configuring the compilation server

Since the compilation server performs caching of dynamic grammars, Nuance recommends that you configure your system to share as few compilation servers as required for redundancy. While the system works with multiple compilation servers, grammars may get compiled by many compilation services and reside in many caches.

If you have a large system, run the compilation servers on a powerful computer with a lot of RAM, and increase the value of parameter `config.EGRCacheMB`.

Parameters for external rule references

The following table describes the parameters that apply to external rule references.

Note: The *compilation-server* executable only uses parameter `config.EGRProxy` and ignores all other parameters. To specify these options when starting the *compilation-server*, use the appropriate *compilation-server* options as described in the *Nuance API Reference*.

Parameter name	Description
`egr.Backoff`	Specifies the global grammar backoff used when an URI reference in a grammar fails.
`config.EGRCacheMB`	Specifies the maximum amount of dynamic grammars to be cached in memory. A larger value will avoid recompiling some `http:` and `file:` grammars but will increase the process size accordingly. Note that the process DOES NOT preallocate the space specified by this parameter. It just accumulates grammars until it hits the maximum amount specified and then starts flushing the least-recently-used ones. Applies to the compilation server and recognition server.
`config.EGRCacheMinFreshSecs`	Specifies the time (in seconds) an `http:` or `file:` grammar is assumed to be "fresh". See "Fetching grammars" on page 30 for more information.
`config.EGRProxy`	If this parameter is set, the compilation server and recognition server do not fetch `http:` grammars directly, but instead fetch them via the specified HTTP proxy server. Valid values are *machinename* or *machinename:port*, for example, `mycomputer:1234`. This parameter is useful if you want all the HTTP requests leaving your site to go through a proxy. Applies to the compilation server and recognition server.
`config.EGRTimeoutMs`	Specifies the maximum number of milliseconds that are allotted for fetching all the external grammar components for a just-in-time recognition. Any requests that are still outstanding once the timeout is reached are reported as load failures with a "Timed out waiting for HTTP response" annotation in the recognition server log. These references are replaced with the backoff reference. See "Specifying a grammar backoff" on page 34 for more information.

Parameter name	Description
`egr.builtin.context`	Specifies the location of the built-in grammars in the system.
`egr.file.context`	Specifies a base `file:` URI; all `file:` URIs in the grammar are relative to this base URI.
`egr.dgdb.context`	Specifies the DBDescriptor for all the `dgdb:` references. This is useful when your application uses the same DBDescriptor for most of the `dgdb:` requests. Set this parameter to the DBDescriptor content, excluding the `dbdesc` keyword. For example: `"provider=fs,root=c:/Temp,class=dgdb,name=Nuance"` The following `dgdb:` references would then fetch the same item: ▪ `<dgdb:?key=foo&dbdesc=provider=oci,server=nuance,name=Nuance,class=dgdb>` ▪ `<dgdb:?key=foo>` ▪ `<dgdb:foo>` ▪ `<dgdb:?key=foo&dbdesc=provider=fs,root=c:/Temp,class=dgdb,name=Nuance>`

Just-in-time grammars

The Nuance System provides *just-in-time* grammars—grammars that let you specify grammar expressions directly as the starting point of a recognition. This section first describes how to write a just-in-time grammar. It then describes how to specify the compilation options required by the recognizer to perform just-in-time compilation.

Writing a just-in-time grammar

A just-in-time grammar is a grammar that lets you specify GSL expressions as well as GSL and GrXML content directly as the starting point of a recognition. A just-in-time grammar does not need to be precompiled. You can define, compile, and use a just-in-time grammar in only one step. You simply pass content directly to the recognizer at runtime through an API call, as shown in the following example:

```
String gsl = "[hello (hi there)]";
jsc.playAndRecognize(gsl);
```

where `jsc` is a *nuance.core.sc.NuanceSpeechChannel*.

The recognition server compiles the grammar "just in time" for the recognition and then discards the grammar after the recognition.

Note that the string passed to the method in the example above is an *extended right-hand side*: it does not contain a name for the rule and does not constitute by itself a complete valid GSL definition. Specifying a GSL fragment only is useful when you do not need to specify a complete set of rules.

You can specify an actual set of GSL or GrXML rules as a just-in-time grammar, as shown in the following example:

```
String gsl = ";GSL2.0 \n" +
   "Rule1:public (please call <http://myhost/people.gsl>)";
jsc.playAndRecognize(gsl);
```

You can also specify a grammar document through an external rule reference. For example, consider the following file:

http://host/grammars/MyGrammar.grxml

```
<?xml version="1.0" ?>
 <grammar xmlns="http://www.w3.org/2001/06/grammar" version="1.0"
       xml:lang="en-US">
 <rule id="main">
 <one-of>
   <item>hi</item>
  <item>bye</item>
 </one-of>
 </rule>
 </grammar>
```

You use this grammar just-in-time by specifying a reference to the file through a recognition API function, as follows:

```
String grxml = "<http://host/grammars/MyGrammar.grxml>";
jsc.playAndRecognize(grxml);
```

Here are more examples:

```
;external reference to a static grammar rule
String gsl = "<static:Names>";
jsc.playAndRecognize(gsl);


;external reference to a dynamic grammar rule
String gsl = "<dgdb:?key=MyGram&dbdesc=provider=fs," +
            "root=/usr/dbs,name=db1, class=dgdb>";
jsc.playAndRecognize(gsl);

;external reference to a GSL file
String gsl = "<file:/c:/nuance/names.gsl>";
jsc.playAndRecognize(gsl);
```

Specifying just-in-time grammar options: Using the keyword RECOGNIZE

You can specify a number of options related to just-in-time grammar compilation and usage with the keyword `RECOGNIZE`. This keyword takes a set of `<key>:<value>` pairs, followed by a blank line, followed by a GSL or GrXML document.

```
RECOGNIZE\n
header1:value\n
header2:value\n
...
\n
(grammar document -- starting with GSL or XML header)
```

Where `\n` represents a newline.

Note: The `RECOGNIZE` keyword must be followed by a grammar document, not an extended right-hand side. See “Creating a grammar document” on page 13.

For example:

```
RECOGNIZE
Nuance-Package-Name: French.1
Nuance-Config-Name: comp-server-french

;GSL2.0
Rule1:public (please call <http://myhost/people.gsl>)
```

Here is an example of a GrXML grammar:

```
RECOGNIZE
Nuance-Package-Name: French.1
Nuance-Config-Name: comp-server-french

<?xml version="1.0" ?>
 <grammar xmlns="http://www.w3.org/2001/06/grammar" version="1.0"
       xml:lang="en-US" root="main">
 <rule id="main" scope="public">
  please call <ruleref uri="http://myhost/people.gsl"/>
 </rule>
 </grammar>
```

Note: Any whitespace on either side of the colon in the `<key>:<value>` pair is ignored.

Nuance supports three keys:

Key	Description
`Nuance-Config-Name`	Specifies the compilation server to use for a just-in-time recognition request. See "Choosing between multiple compilation servers" on page 43.
`Nuance-Package-Name`	Specifies the master package to use for a just-in-time recognition request. See "Multi-package support" on page 44.
`Content-Base`	Specifies a base URI; the relative URIs in the grammar are relative to this base URI. See "Specifying the base for relative URIs" on page 45.
`Nuance-Grammar-Label`	Specifies a label for the recognition request. You should set this label if you want to generate analysis and tuning reports for just-in-time grammars. Set this key to a meaningful value, for example the name of the current recognition state: `Nuance-Grammar-Label: GetPizzaSize` See "Organizing the input data into groups" on page 144 for more information.

Choosing between multiple compilation servers

By default, when you use a just-in-time grammar, each external grammar reference is sent to any available compilation server. However, some applications may need to work with multiple, differently configured compilation servers. For example:

- You need to turn optimization off for some grammars (using parameter `comp.Optimize`) because they would take too long to compile, but you want to keep it on for other grammars.
- You want to set the number of pronunciations generated (using parameter `pron.MaxProns`) to a small number for large grammars, but to a large number for small grammars.

In cases like this, you can run separate compilation servers, each with the appropriate compilation options.

To run a system with multiple compilation servers:

1 Specify a name for each server you start using the *-config_name* argument. For example:

```
> compilation-server -package English.America.1
pron.MaxProns=10 -config_name SmallGramServer
> compilation-server -package English.America.1 pron.MaxProns=3
-config_name LargeGramServer
```

These commands start two compilation servers, *SmallGramServer* and *LargeGramServer*.

2 When you call a recognition function with a just-in-time grammar, specify the name of the compilation server using the `Nuance-Config-Name` key-value pair in the `RECOGNIZE` header of your just-in-time grammar specification, as follows:

```
String gsl = "RECOGNIZE\n" +
      "Nuance-Config-Name: LargeGramServer\n" +
      " \n" +
      ";GSL2.0" +
      "PEOPLE:public <http://server/people.gsl>";
jsc.playAndRecognize(gsl);
```

In this example, grammar *people.gsl* will be compiled using the *LargeGramServer* compilation server, which only generates three pronunciations.

Note: The words in the just-in-time grammar itself are compiled by the recognition server, not the compilation server. For example, in the following grammar, the words "hello”, “hi”, and “there" are compiled by the recognition server. Therefore, in such a case, the GSL option `Nuance-Config-Name` key-value pair is ignored.

```
String gsl = ";GSL2.0 \n" +
       "[hello (hi there)]";
jsc.playAndRecognize(gsl);
```

Multi-package support

You can use just-in-time grammars with multiple Nuance master packages. To do so, you compile your grammar with the different master packages and specify which master package to use for a specific recognition.

To use a just-in-time grammar with multiple packages:

1 Use *nuance-compile* to compile a grammar with the different master packages. For example, consider an application that needs to perform recognition both in French and English using the same grammar. You compile the grammar as follows:

```
> nuance-compile MyEnglish.grammar English.America.1
-enable_jit
> nuance-compile MyFrench.grammar French.1 -enable_jit
```

These commands will create two recognition packages, *MyEnglish* and *MyFrench*. If necessary, you can override the package names with the *-package_name* option, as follows:

```
> nuance-compile MyEnglish.grammar English.America.1
-enable_jit -package_name MyEnglishPackage
> nuance-compile MyFrench.grammar French.1 -enable_jit
-package_name MyFrenchPackage
```

2 When you perform recognition, specify the package name to use by setting the `Nuance-Package-Name` key-value pair in the `RECOGNIZE` header of the just-in-time specification. For example:

```
String gsl = "RECOGNIZE\n" +
      "Nuance-Package-Name: French.1 \n" +
      " \n" +
      ";GSL2.0" +
      "Greetings:public [salut bonjour]";
jsc.playAndRecognize(gsl);
```

If you have overwritten the default package name, you must specify the new package name, as follows:

```
String gsl = "RECOGNIZE\n" +
      "Nuance-Package-Name: MyFrenchPackage \n" +
      " \n" +
      ";GSL2.0" +
      "Greetings:public [salut bonjour]";
jsc.playAndRecognize(gsl);
```

Specifying the base for relative URIs

The `Content-Base` key lets you use relative URIs in your grammar document. The relative URIs in the grammar are relative to the base URI specified in the `Content-Base` key.

Consider the following grammar:

C:\tmp\head.gsl

```
RECOGNIZE
Content-Base: http://yahoo.com/grammars/stocks/

;GSL2.0
Rule1:public (please call <../people/work.gsl>)
```

The `<../people/work.gsl>` URI will be expanded to `<http://yahoo.com/people/work.gsl>`.

Note that any character following the final "/" character in the `Content-Base` value is ignored. For example, the following specification is the same as the specification in the example above:

```
Content-Base : http://yahoo.com/grammars/stocks/abcdef
```

Redefining an external rule to a local rule

The standard GSL syntax as specified in this guide also applies to just-in-time grammars. You can also use the keyword `local:`, available in just-in-time grammars only. This keyword lets you redefine an external rule to one that is defined locally.

For example, consider the following file:

file:/c:/usr/local/template.gsl

```
;GSL2.0
MAIN_MENU:public [ (get me a quote for STOCK)
             help
             cancel
             OTHER_CMDS ]
OTHER_CMDS:dynamic []
STOCK:dynamic []
```

The following just-in-time grammar redefines the *OTHER_CMDS* rule to a local one:

```
String gsl = ";GSL2.0 \n" +
   "MY_MAIN_MENU:public [<<file:/usr/local/template.gsl> ! \n" +
         " OTHER_CMDS = <local:CMDS> & \n" +
         " STOCK = <http://stocks.com/list.gsl>>] \n" +
         " CMDS:public [up down left right] \n" +
         "])";
jsc.playAndRecognize(gsl);
```

Note that, as with other references, the rule that you are redefining locally must be a public rule.

Enabling compilation of just-in-time grammars

To compile a recognition package to use with a just-in-time grammar, run *nuance-compile* with the *-enable_jit* option, as follows:

```
> nuance-compile my.grammar English.America.1 -enable_jit
```

In this example, the *English.America.1* package can be used to perform recognition of a just-in-time grammar. If you do not specify the *-enable_jit* option, your application can only perform recognition using the package's static grammars and contexts.

See "Compilation of just-in-time grammars" on page 100 for more information.

Note that when referring to an NGO grammar in a JIT grammar, the NGO grammar and the JIT grammar must have been compiled using the same master package. Otherwise, the reference to the NGO grammar will be treated as a `fail:` reference.

Usage scenarios

This section presents typical situations in which using just-in-time grammars allows a more efficient and simpler implementation of an otherwise complex piece of code.

Case 1: A personalized bill pay application

Consider a bank that lets its customers pay bills using a voice-recognition application. Each customer has a personalized list of a few dozen names. These lists are stored in a set of files on an internal web server; for example:

```
http://megabank.com/payees/customer1.gsl
http://megabank.com/payees/customer2.gsl
...
```

The contents of these files might look like this:

```
;GSL2.0
MyPayees:public [   (?my visa card)
                    (?the mortgage)
                     .....
                    ]
```

When a customer calls, for example customer #1, the application invokes recognition with a just-in-time grammar, as follows:

```
;GSL2.0
ROOT:public ([(i ?really want to pay)
          (?please pay)]
         <http://megabank.com/payees/customer1.gsl>)
```

Variant 1: Using a precompiled grammar

In the example above, each payee list was stored as a GSL file on a web server. To load the files to the recognition server more quickly, you could also run *nuance-compile-ngo* on each file and refer to the *.ngo* file instead. The GSL and NGO files can also be referenced using `file:` URIs.

Variant 2: Using a dynamic grammar database

Using the `newDynamicGrammar` functions or the *dgdb-edit* utility, you could also store the compiled grammars in a dynamic grammar database, and point to the grammars using `dgdb:` URIs. Nuance recommends this approach if the company deploying the application requires the reliability, redundancy, and scaling associated with a commercial database.

Variant 3: Using speaker-trained grammars

If customers can add names to their list using voice enrollment—by speaking the names to add—you could store these grammars in a dynamic grammar database. At runtime, you point to the appropriate grammars using the `dgdb:` URI.

Case 2: Single-use GSL

Consider an application that deals with ambiguous responses from users by constructing a grammar with these ambiguous responses, playing a prompt to the caller such as "Did you mean ...?", and performing recognition on that grammar.

Without just-in-time compilation, you would:

- Create a static grammar that includes a reference to a dynamic grammar and compile the static grammar
- At runtime, compile the GSL code dynamically and store in a dynamic grammar database
- Insert the dynamic grammar into a static grammar
- Recognize using that static grammar
- Delete the dynamic grammar

In contrast, using a just-in-time grammar, you would only use a code fragment like the following:

```
String gsl = "[(Cisco Systems) (Sysco Foods)]";
jsc.playAndRecognize(gsl);
```

Case 3: A VoiceXML interpreter

A VoiceXML interpreter needs to recognize speech using all in-scope grammars in parallel. For instance, consider the following VoiceXML fragment (used for illustration purposes only):

http://host/vxml/main.vxml

```
<vxml>
   <link next="Navigation.vxml#GoForward">
      <grammar src="..\grammars\GoForward.gsl" />
   </link>
   <link next="Navigation.vxml#GoBack">
      <grammar type="application/x-nuance-gsl">
         (go back)
      </grammar>
   </link>
   <form id="mainmenu">
      <field name="mainmenu" slot="command">
         <grammar src="..\src\grams\roomorama2.gsl" />
      </field>
   </form>
</vxml>
```

Without just-in-time compilation, you would need to write fairly complex code to implement this logic. Recognizing with a just-in-time grammar is far simpler. You would write code to assemble these three grammars into one grammar and use it as follow:

```
String gsl = "[<http://host/grammars/GoForward.gsl>\n" +
               "(go back)\n" +
                "http://host/src/grams/roomorama2.gsl ]";
jsc.playAndRecognize(gsl);
```

Using the dynamic grammar gate technique

You can use dynamic grammars to dynamically enable or disable various branches in an otherwise static grammar, using what Nuance calls the *gate* technique. When you use the gate technique, you can explicitly “open” or “close” each branch of the grammar. You can also use gate *resistors* to dynamically change the probability associated with specific branches.

For example, in a personalized brokerage application you might only want to allow users to select a stock to sell if their portfolio actually includes that stock. When a given user wants to sell a stock, you insert a dynamic grammar into each of the placeholders that either opens the gate (allowing the grammar to be recognized) or closes the gate (disabling the grammar), based on whether the current user owns each stock:

- To open a gate, refer to a dynamic grammar containing the following keyword:

```
special:passthrough
```

- To close a gate, refer to a dynamic grammar containing the following keyword:

```
special:roadblock
```

These expressions function as gates for the following reasons:

- An `AND` construction represents a set of expressions that must be matched. The keyword `special:passthrough` is an empty `AND` construction, which means that nothing needs to be matched. The gate is therefore "open"—the recognizer continues trying to match the other phrases in the larger grammar.
- An `OR` construction represents a list of alternatives, one of which must be matched. The keyword `special:roadblock` is an empty `OR` construction. It provides no alternatives and therefore a match is impossible. The gate is "closed"—the recognizer stops trying to match the utterance against the phrases in this grammar.

Note: In a Nuance GSL grammar, you can also use the `NULL` and `VOID` special keywords, as defined by W3C grammars: `NULL` and `passthrough` are synonyms; `VOID` and `roadblock` are synonyms.

The syntax of the special reference is shown in the following example:

```
;GSL2.0
TEST_SENTENCE:public
   [
      (the quick <special:passthrough> brown fox)
      (the sentence <special:roadblock> that cannot be spoken)
   ]
```

In this example, the first sentence—"the quick brown fox"—can be recognized while the second sentence—"the sentence that cannot be spoken"—will never be matched.

To use dynamic grammar gates in an application, you can create a static grammar with dynamic grammar placeholders. At runtime, you can redefine the placeholders to the "open" dynamic grammar or "closed" dynamic grammar, as appropriate for the current user.

You can also use a *resistor* instead of an open or closed gate grammar. This lets you change the probability that any given branch will be taken instead of simply disabling the branch. To do this, insert a dynamic grammar record containing they following keyword:

```
<special:resistor?weight=XYZ>
```

where XYZ is the probability for the branch. For example:

```
<special:resistor?weight=0.5>
```

Probabilities that are less than 1.0 diminish the chance that the branch will be parsed. The closer the probability gets to 0, the less likely the branch is to be recognized. Specifying a probability of 0 is the same as closing a gate, and specifying a probability of 1 is the same as opening a gate. You can also assign probabilities that are greater than 1.0, for example:

```
<special:resistor?weight=1.5>
```

These cause the recognizer to favor that branch rather than avoid it. You might use this, for example, in a stock quote system where you want to boost the likelihood of recognizing a stock in the current user's portfolio without disallowing other stocks.

GSL syntax for enrollment

Enrollment is the feature that lets users add spoken phrases to a dynamic grammar. During enrollment, the recognition system listens to the user speak a phrase, generates pronunciation for that phrase by performing phonetic recognition of the utterance, and returns the pronunciation, along with a phrase identifier, to the application.

To perform voice enrollment, the recognition package of your application must include the special grammar *EnrollmentGrammar*, defined in the Nuance grammar file *enrollment.grammar*. This grammar enables phonetic recognition, and allows the system to generate pronunciations for spoken phrases. Add this as a subgrammar in a top-level grammar, for example:

```
; enrollment.grammar defines "EnrollmentGrammar"
#include "enrollment.grammar"
.PersonalPayeeList EnrollmentGrammar
```

When you perform enrollment, you use this as the recognition grammar. This grammar can also include any words or subgrammars needed for your application. For example, your application might allow the user to say things like "help" and "cancel" during enrollment, in which case your enrollment grammar should include those phrases. You compile this grammar just like any other grammar. The resulting recognition package can be used for both standard (non-enrollment) recognition tasks, or to generate the pronunciations for utterances.

Note that the enrollment grammar is what you specify only during the enrollment process. To use this grammar for recognition, you still have to insert it into a static grammar, at a location defined by the syntax:

```
GrammarName:dynamic []
```

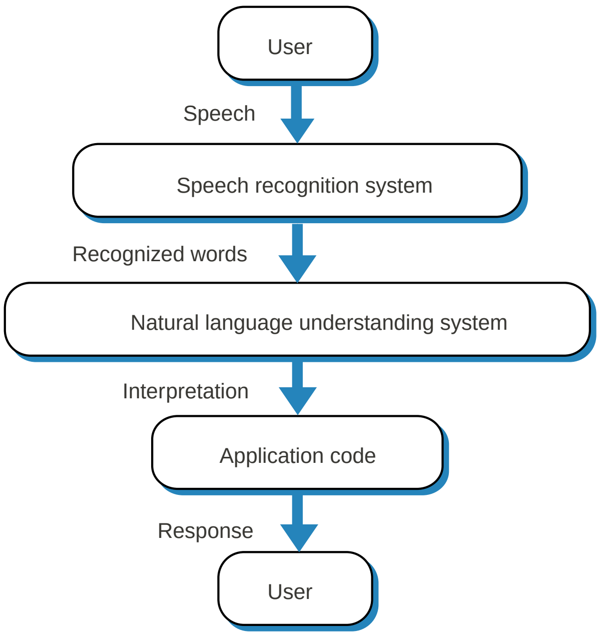
For more information on enrollment see the *Nuance Application Developer's Guide*.

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 zeros 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 the "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 *$return* variable

All rule references are automatically assigned to the magic variable *$return*, unless an explicit variable assignment like the following is present in the file:

```
SomeGrammarName:aVariableName
```

$return is a read-only variable that refers to the return value of the last rule reference in a given rule expansion.

Here are a couple of fragment specifications, in GSL and GrXML, illustrating the use of the *$return* magic variable.

GSL

```
Main    (Command Number:n shares)
                  {<command $return> <num $n>}

Command [Buy Sell]      {return($return)}

Buy [buy purchase]      {return(buy)}
Sell [sell]            {return(sell)}
Number ...
```

GrXML

```
<rule id="Command">
   <one-of tag="return($return)">
      <item>
         <ruleref uri="#Buy"/>
      </item>
      <item>
         <ruleref uri="#Sell"/>
      </item>
   </one-of>
</rule>
```

The assign command

To copy the value of a *$return* variable to a specific variable, Nuance introduced the `assign` command.

This is a general purpose command that takes a destination variable as its first argument and any valid r-value—such as, a *$variable* or a command—as its second argument, assigning the computed value of the second to the first.

Here are a couple of fragment specifications, in GSL and GrXML, illustrating the use of the `assign` directive.

GSL

```
Main    (First:f Second:s)
      {assign(v add($f $s)) return(mul($v 2))}
```

GrXML

```
...
<rule id="Main">
   <item tag="&lt;command $v&gt; &lt;num $n&gt;">
      <ruleref uri="#Command" tag="assign(v $return)"/>
      <ruleref uri="#Number" tag="assign(n $return)"/>
   </item>
   please
</rule>
...
```

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

The slot definitions file is a plain text file that lists all the slot names used in your grammars, one per line. This file must contain at least one slot name.

Using the command-line utilities, you need to manually create the slot definitions file and 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.

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.

Compiling a grammar with natural language support

The grammar compilation program *nuance-compile*, described in Chapter 6, 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 weights and probabilities” on page 20 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
- Subdialogs

To test your grammar for ambiguity from the command line, use the program *generate*, described in “Command-line tools for grammar testing” on page 116.

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 54. 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 *TwoDigit* grammar 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 4: 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 *ThreeDigit* grammar 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))}
```

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 *DateConstraint* grammar 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: 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 5: 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 6: 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.

Semantic uniqueing

In general, before a set of answers is returned during N-best processing they undergo a process known as “semantic uniqueing.” This means that if multiple answers have the same natural language interpretation, only one of those answers is returned. If the recognition engine identifies two word strings with the same interpretation, then only the highest-scoring result is returned in the N-best list. As a consequence, you are guaranteed that all the answers in your N-best list differ semantically.

Semantic uniqueing is performed unless you have turned off interpretation by setting `rec.Interpret` to `FALSE`

There is one scenario you should be aware of. If you have no natural language within a particular top-level grammar, presumably you do not want semantic uniqueing to be performed with that grammar. However, if another top-level grammar in your package does include natural language definitions, semantic uniqueing is performed for all grammars (assuming `rec.Interpret` is `TRUE`). A future release may include the ability to explicitly disable semantic uniqueing.

Robust natural language engine

Nuance 8.5 offers a robust NL parsing capability that lets you write slot-filling grammars for the meaningful phrases *only*. The NL interpretation engine spots these meaningful phrases in the text output of the recognizer and fills the appropriate slots. This technology is best used with SLM grammars. For more information about robust NL interpretation, see “Robust natural language interpretation” on page 88.

Chapter 5

Say Anything: Statistical language models and robust interpretation

5

The Say Anything™ feature, which includes Nuance's statistical language models (SLM) and robust natural language interpretation (robust NL) technologies, allows users to speak freely to an application and have their sentences interpreted by the NL engine without having to write complex grammar rules covering the entire sentence.

This chapter describes the SLM technology and explains how the robust interpretation of spoken sentences is performed. It then describes how to create an SLM grammar and how to use robust NL interpretation.

Overview

This section describes the motivation behind the Say Anything feature and explains how it can improve the accuracy of an application without the need to write complex GSL grammar rules.

The challenge

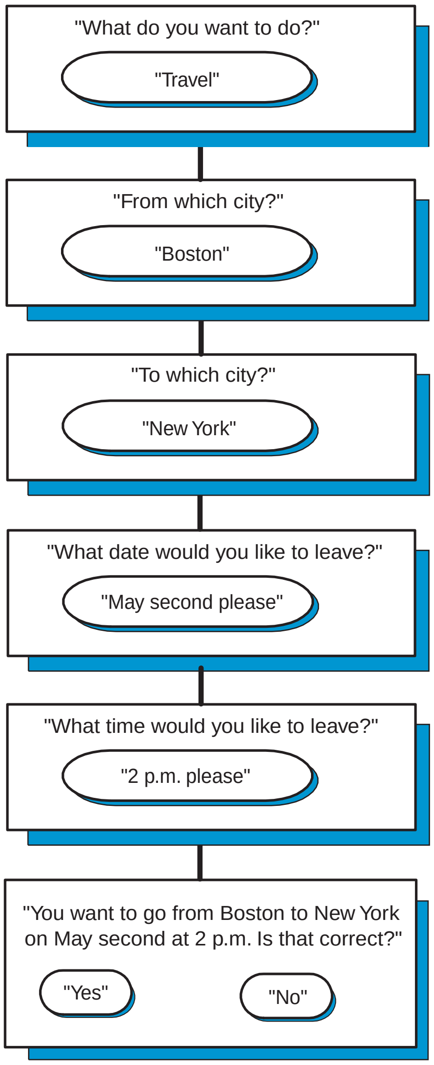
Consider an application that requires an open-ended prompt such as "What can I help you with?". Not only can the user's response be highly variable and hard to predict, but it can also contain *disfluencies*—things such as re-starts, filled pauses ("*um*" and "*uh*"), and ungrammatical sentences. In addition, the grammar for this application may need to fill several NL slots from only one utterance.

The challenge is to write a GSL grammar that lets callers say any arbitrary phrase within the domain of the task, fills many slots, and still achieves high accuracy.

Coming up with such a set of grammar rules can be tedious. In most cases, the out-of-grammar rate obtained with hand-crafted GSL rules is very high and any attempt to add more GSL rules often leads to poor in-grammar recognition accuracy.

A simple solution

One attempt to solve this problem is to design a dialog in the form of nested sets of menus. For example, consider an application for airline ticket reservations. A dialog that used nested sets of menus would look like the following:


"What do you want to do?"
"Travel"
"From which city?"
"Boston"
"To which city?"
"New York"
"What date would you like to leave?"
"May second please"
"What time would you like to leave?"
"2 p.m. please"
"You want to go from Boston to New York on May second at 2 p.m. Is that correct?"
"Yes"
"No"

While this approach works well for certain applications (for example, a simple StockQuote application), it may not be suited for applications that must request a lot of information from callers. Users may find it difficult and frustrating to navigate through such complex dialogs.

An alternative approach is to construct a grammar such as:

```
.Sentence +Vocab
```

where *Vocab* is a subgrammar consisting of the list of all relevant words in the task domain:

```
Vocab [word1 word2 ... wordN]
```

For example, in a flight reservation application, the subgrammar would resemble:

```
Vocab [a Boston direct flight (san francisco) from i is next (new
york) ninth on sixth the there to travel uh um want]
```

Suppose that the number of words in the vocabulary is N. Since every word in this vocabulary can follow every other word with the same likelihood, such an open-ended grammar usually performs poorly when N is large enough to cover most relevant words in the domain, typically several thousand.

The performance of the above grammar can be improved by assigning probabilities to the words in the vocabulary, based on the likelihood of their occurrence in a typical sentence. These probabilities can be evaluated by collecting a set of training examples and estimating the frequencies of words. In fact, Nuance provides a tool, *compute-grammar-probs*, that performs this task automatically.

However, a list of words with probabilities—commonly referred to as a *unigram*—is an overly simplistic model for many reasons, mostly because the probability of a word is *independent* of its position in a sentence. In this model, for example, a sentence such as *I want to travel to Boston next Monday* is considered as likely as any permutation of its words, such as *To want to Boston I travel next Monday*.

A better solution: The SLM and the robust interpretation approach

A language model that assigns probabilities to sequences of words is called a statistical language model (SLM). The unigram model described above is an example of a very simple SLM. Using *n-gram* SLMs provides a more interesting solution.

An n-gram SLM is one in which the probability of a word depends on the previous N-1 words. N is called the *order* of the model. The unigram model described above is an example of an n-gram model of order 1. In a unigram, the probability of a word is not affected by the context preceding that word. A 2-gram SLM—often called a *bigram*—is a model in which the probability of a word changes according to the word that precedes it. Hence, a bigram with a vocabulary of size V contains V^2 probabilities. Similarly, an SLM of order 3—often called a *trigram*—assigns V^3 probabilities.

Unlike a GSL grammar, an SLM grammar is not manually written but *trained* from a set of examples that models the user's speech. To train an SLM grammar, you pass a set of examples (and optionally a domain-specific vocabulary) to a Nuance utility, which estimates the model probabilities.

Since the probability assigned to a phrase depends on the context and is driven by data, an SLM provides a grammar in which more plausible phrases are given higher probabilities, a feature that the unigram model described above supports in a limited way only.

SLMs are useful for recognizing free-style speech, especially when the out-of-grammar rate is high. SLMs are not meant to replace GSL grammars, which are quite suitable when the application's prompts are sufficient to restrict the user's response. Since SLMs need a large set of examples to train, a data collection system or a pilot based on a GSL grammar may need to be developed to gather training examples.

Creating an SLM grammar

To create an SLM grammar, you:

- Create the training set
- Create a vocabulary file (optional)
- Determine the order of the model
- Train the SLM grammar
- Use the grammar in your application
- Measure the perplexity of a model

These steps are described below. The first sections describe the procedure in detail and provide conceptual information you need to perform the procedure. For a summary of the procedure, see "Summary of the procedure for creating an SLM grammar" on page 87.

Create the training set

The first step in creating an SLM grammar is to create the *training set*. A training *set* is a text file consisting of transcriptions of sample sentences, one per line, that users can say. Just as with any GSL grammar, the words cannot contain any uppercase letters, since *nuance-compile* assumes that uppercase words are rule names. The following example shows an excerpt of a training set for a flight reservation application:

```
um friday june the ninth uh from new york to austin
i want a flight from san francisco to los angeles on july sixth
is there a direct flight from boston to kalamazoo
i want to travel next monday
```

Your training set should not include punctuation and special characters. Also, all words and abbreviations should be written out the way they will be spoken. For example, *January 23* should be written *january twenty third* and *St. Patrick St.* should be *saint patrick street*.

There is a strong correlation between the quality of an SLM and the quality of the training set. The best training sets are those collected from actual users of a system similar to what is to be deployed. You need several thousand examples at a minimum. For example, for vocabulary sizes up to 2,500 words, about 20,000 training examples are adequate to properly train an SLM.

Note that you may also want to create another file of sample sentences to use in your test set, to measure the quality of your model (as described in "Measure the perplexity of a model" on page 86).

Overview of class-based SLMs

The phrases in a training set can contain grammar rules; each grammar rule is called a *class*. An SLM that contains grammar rules is called a *class-based* SLM. For each class, you either write a GSL grammar rule that generates the phrases in the class or train a separate SLM. This procedure is described in the next section.

Class-based language models are useful when you have a limited amount of training data. In that case, certain important words or phrases may be rarely seen and therefore have poorly estimated statistics. A class lets you group similar words or phrases. For example, the class *City* in the above training set will probably be seen in your training data more often than individual city names, so its statistics will be better estimated and thus yield a better language model.

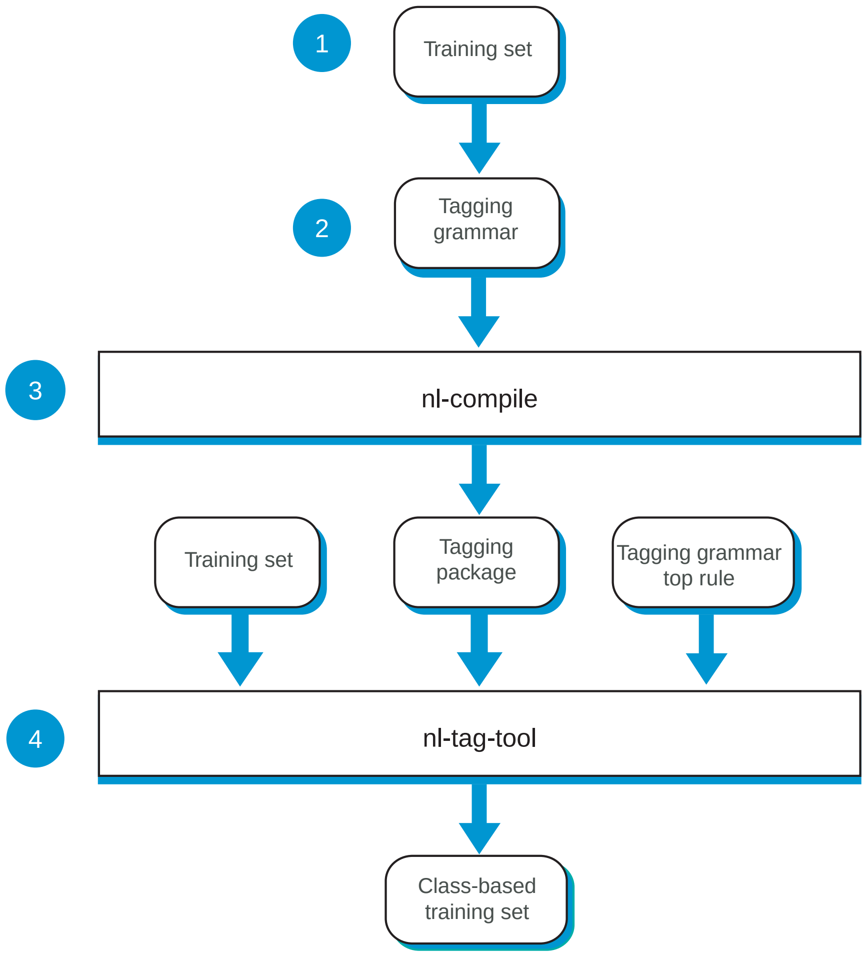
For example, consider the training set described above. You could generalize these examples by replacing the city names by the class name *City*. The *City* grammar rule would need to be defined as a separate GSL or SLM.

Class-based SLMs also let you leverage the existing standard grammars developed by Nuance for complex concepts such as time, date, and money amount. Furthermore, class-based language models let you dynamically modify the SLM vocabulary. For example, if the *City* class is a dynamic grammar, your application can dynamically change the entries in the *City* subgrammar at runtime, without retraining or recompiling the SLM.

Creating a class-based training set

To create a class-based SLM, you use the *nl-tag-tool* utility, which replaces phrases in a training set by the names of grammar rules. This process is known as *tagging*.

The *nl-tag-tool* utility works as follows. First, you look at your training set to determine the appropriate classes (step 1). You then write a grammar for these rules, called a *tagging grammar* (step 2). You compile this grammar into a tagging package (step 3). You then call *nl-tag-tool* with this tagging package, the training set, and the top rule of the tagging grammar. This utility converts the training set into a class-based one by processing the sentences and tagging them according to the grammar definitions. The following figure illustrates this process.



The *nl-tag-tool* utility takes sentences from standard input—you can enter sentences one at a time and see the resulting interpretation(s). You can also use *nl-tag-tool* in batch mode, as shown above, by creating a file with a list of sentences and using input redirection.

Note: *nl-tag-tool* also lets you filter which rules in a grammar should be tagged. See the *Nuance API Reference* for more information about this option and for a detailed description of the *nl-tag-tool* utility.

To create a class-based SLM:

1 Look at your training set to determine the appropriate class rules.

For example, in the training set described above, you could create the following classes: *City*, *DayOfWeek*, and *Month*.

2 Write a GSL grammar that defines each of these rules. This grammar, called the *tagging grammar*, will be used to relabel the training set.

For example, the following grammar, called *flightinfo.tagging.grammar*, defines the *City*, *DayOfWeek*, and *Month* rules:

```
.Tagging [ City DayOfWeek Month ]

City [ austin~0.15 boston~0.2 kalamazoo~0.05 (los angeles)~0.2
(new york)~0.2 (san francisco)~0.2]

DayOfWeek [ sunday monday tuesday wednesday thursday friday
saturday ]

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

Note: If you have statistically valid information about the probabilities of the words in your application, Nuance recommends that you use these probabilities. For details on this topic, see "Using weights with special operators" on page 21.

3 Create a tagging package using *nl-compile*.

For example, the following command compiles the *flightinfo.tagging.grammar* grammar:

```
> nuance-compile flightinfo.tagging.grammar English.America
```

Use a simple master package like *English.America* to produce the tagging package. This package will not be used for recognition.

4 Run *nl-tag-tool* to tag the training set, as follows:

```
> nl-tag-tool -package flightinfo.tagging -grammar .Tagging
-no_output < training.txt > training.tagged.txt
```

This command takes the current training set (for example, *training.txt*) and relabels it according to the package and grammar specified (for example, the *flightinfo.tagging* package and the *Tagging* grammar). The new training set is saved in file *training.relabeled.txt*.

For example, the relabeled training set would look as follows:

```
um DayOfWeek Month the ninth uh from City to City
i want a flight from City to City on Month sixth
is there a direct flight from City to City
i want to travel next DayOfWeek
```

Create a vocabulary file (optional)

You can also create a file that contains the *vocabulary* of your grammar—all the words to be covered by the SLM. This option is useful to constrain the vocabulary if the examples include typos, noise markers, or data from other applications containing non-relevant words that you do not want to include in your vocabulary.

This list of words is kept in a text file, one word per line. You need to split each utterance and phrase into unique words. For example, an utterance like “New York” needs to be split over two lines. Again, all the vocabulary words cannot contain any uppercase letters, since *nuance-compile* assumes that uppercase words are rule names.

The following is an excerpt of a vocabulary file for the flight reservation application example described above:

```
a
direct
flight
francisco
from
i
is
next
new
ninth
on
san
sixth
the
there
to
travel
uh
um
```

```
want
york
```

If you are using a class-based SLM, you need to determine the vocabulary from the relabeled training set. You also need to include the rule names. For example:

```
City
DayOfWeek
Month
a
direct
flight
from
i
is
next
ninth
on
sixth
the
there
to
travel
uh
um
want
```

Do not include words that are meaningless for the application and that occur once in the relabeled training set (for example, vulgar expressions).

While Nuance recommends that you use a vocabulary file, it is not required. If you don’t specify a file, all unique words in the training set are included in the vocabulary.

Open and closed vocabulary SLMs

There are two types of vocabularies: an open vocabulary SLM and a closed vocabulary SLM:

- An *open* vocabulary SLM is an SLM that contains a special class, called unknown (UNK), that lets you add new words to an SLM without retraining the model.
- A *closed* vocabulary SLM is an SLM to which you cannot add new words.

By default, an open vocabulary SLM is trained.

The UNK class has the following properties:

- If you have a vocabulary file, all the words in the training set that are not included in the vocabulary are tagged as belonging to the UNK class before the training process begins.

- If you do not have a vocabulary file or if all the words in the training set are in the vocabulary file, the UNK class is still generated but is assigned a small probability weight.

You need to define the UNK class in the GSL grammar that will include the SLM grammar. You can define it as an empty subgrammar or turn it into a background noise model for robustness, using items such as `@reject@`. You can also add new words to the UNK class. Nuance recommends that you use probabilities for all items defined or added to the UNK class.

Note that using the UNK class to add new words to an SLM should be a temporary solution only. Ideally, you should get new training samples that include the new words and then repeat the SLM training process to obtain proper statistics on the new words.

You can direct *train-slm* to train a closed vocabulary SLM—that is, with no UNK class—by passing the command-line option *-no-unknown*.

Determine the order of the model

The next step is to determine the order of the model. Is a trigram SLM better than a bigram for your application? It is correct to assume that the larger the N, the more powerful the n-gram model is since a larger context is used to assign a probability to a word. However, the number of probabilities needed in the model grows as a power of N and are therefore more difficult to estimate at large N values. Also, by increasing N when the training samples are limited, the model may experience over-training, that is, the model has memorized the training set and hence has lost its ability to model sentences not covered in the training set.

The optimum value of N is usually empirically determined by training a number of different n-gram SLMs and measuring their performance by running *batchrec* experiments on a set of example sentences (at least 500 sentences are needed to get a reasonable estimate of the performance). Your *batchrec* test set must not include sample sentences that were used to train your SLM. For more information on running batchrec to verify the performance of recognition packages, see Chapter 8, “Testing recognition performance.”

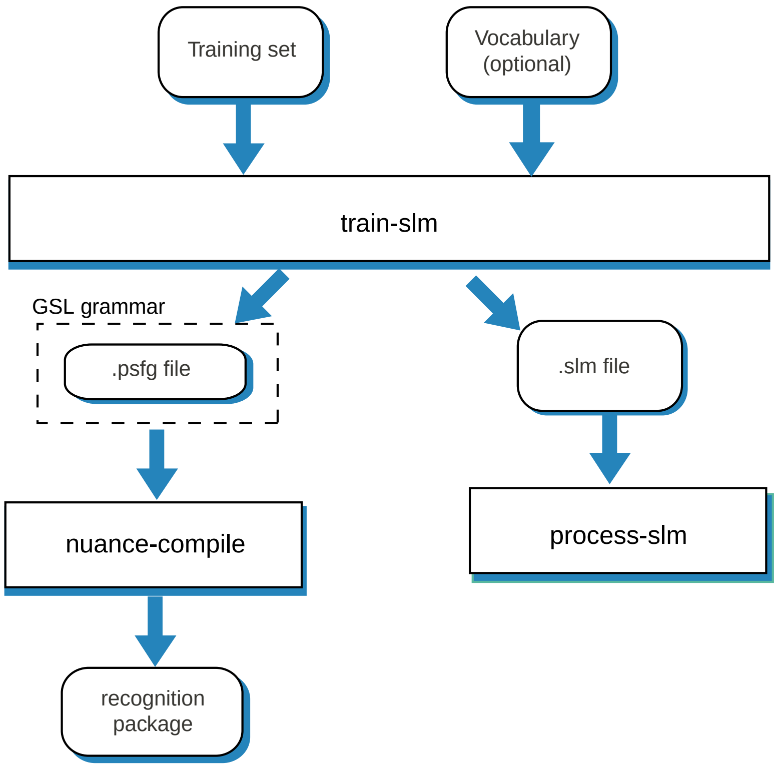
You can also use perplexity measurements, as described in “Measure the perplexity of a model” on page 86, to determine the value of N. However, *batchrec* is the preferred method.

Train the SLM grammar

The next step is to train the SLM grammar. To train an SLM, you use the *train-slm* tool, which works as follows. It takes as input a training set and optionally a vocabulary file, and it generates two output files that share the same base name but have different extensions:

- A *.pfsg* file, which is a format that can be processed by the Nuance speech recognition engine. This is the file that you will be using in your Nuance application.
- An *.slm* file, which contains the basic probability values for the SLM. This file can be used for perplexity measurement, as described in “Measure the perplexity of a model” on page 86. Advanced users can also use this file for further manipulation. See Appendix B, “Advanced SLM features” for more information.

The following figure summarizes the training process.



Procedure

To train an SLM, enter the following command:

```
train-slm -corpus training-set-filename
         -slm output-slm-basename
         [-vocab vocab-filename] [-n N] [-no-unknown]
```

Where:

- `training-set-filename` is the name of the file containing the training set. See “Create the training set” on page 75 for more information on creating the training set.
- `output-slm-basename` is the base name of the output *.pfsg* and *.slm* files created by the trainer. The files are saved in the current directory.
- `vocab-filename` is the name of the optional vocabulary file. If you don’t specify this option, all unique words in the `training-set-filename` are included in the vocabulary. See “Create a vocabulary file (optional)” on page 79 for more information on creating the vocabulary.
- `N` is the order of the SLM to be generated. The default value is 3, a trigram SLM.
- `-no-unknown` trains a closed vocabulary SLM.

For example, to train a class-based SLM grammar for the flight reservation application with an open vocabulary SLM, enter the following command:

```
> train-slm -corpus training.relabeled.txt -vocab vocab -slm
flightinfo
```

The *train-slm* tool takes the class-based *training.relabeled.txt* training set and the *vocab* vocabulary file and generates the following two files:

- *flightinfo.slm*
- *flightinfo.pfsg*

The SLM will be a trigram, open vocabulary SLM.

Use the grammar in your application

You are now ready to use the SLM grammar in your application. To use an SLM grammar, you include the *.pfsg* file in a GSL grammar file using the following GSL rule:

Note: The `:slm` syntax replaces the Nuance 7.0 `:lm` syntax.

```
SLMSubGrammar:slm "output-slm-basename.pfsg"
```

Where:

- `SLMSubGrammar` is a subgrammar created from the *.pfsg* file. This grammar is treated like any other GSL subgrammar and can be included in other GSL grammars.
- `output-slm-basename.pfsg` is a *.pfsg* file created by *train-slm*.

  Note: You cannot specify the *.slm* file instead of the *.pfsg* file.

For example, to use the SLM grammar created for the flight reservation application, you would create the following grammar, called *flightinfo.grammar*:

```
.Top SLMRule
SLMRule:slm "flightinfo.pfsg"
#include "flightinfo.tagging.grammar"
UNK [ @reject@ ]
```

You then compile the GSL grammar using *nuance-compile*, as follows:

```
> nuance-compile Your_GSL_Grammar Your_Master_Package -auto_pron
-dont_flatten
```

For example:

```
> nuance-compile flightinfo.grammar English.America.1 -auto_pron
-dont_flatten
```

Compilation of large vocabulary SLMs—especially class-based language models—can be slow and require large amounts of memory. Therefore, when compiling a grammar containing statistical models, Nuance recommends that you:

- Use the *nuance-compile* option *-dont_flatten.*
- If you find that compiling your SLM package takes too long or takes too much memory, use the *nuance-compile* option *-dont_optimize_graph* to disable the compilation optimization pass.

Once the speech recognition system starts processing a sentence using the *SLMRule* grammar, it can handle any phrase within the vocabulary of the model.

Using a class-based SLM

If you use a class-based SLM, you need to define the classes in the GSL grammar. For example, the sample grammar above includes the *flightinfo.tagging.grammar* grammar, which defines the *City*, *DayOfWeek*, and *Month*.

Using an open vocabulary SLM

If you are using an open vocabulary SLM, you need to define the UNK class in the GSL grammar. You can define it as an empty subgrammar or turn it into a background noise model for robustness—using items such as @reject@, as shown in the example above. You can also add new words to the UNK class. Remember to use probabilities for all items defined or added to the UNK class.

Final notes

Keep in mind that:

- When statically compiling a grammar containing an SLM rule, *nuance-compile* first looks in the current working directory for the *.pfsg* file. If it does not find the file, it then looks in directory *%NUANCE%\data\lang\language\grammars*.

- An SLM grammar must be a subgrammar in another top-level grammar; it cannot be a top-level grammar itself. However, the grammar can be as simple as:

```
.Top SLMRule
SLMRule:slm "flightinfo.pfsg"
```

Set appropriate parameters

Some recognition parameters significantly affect the recognition performance in SLM-based applications and should be set appropriately:

1 Set `rec.GrammarWeight` to a value of 9 or higher.

This parameter controls the weight of the probabilities in the grammar or statistical model, versus the acoustic knowledge.

Since all word strings in an SLM are valid utterances, it is important to help the recognizer prune unlikely word sequences based on the probabilities. The default value of 5 for this parameter is too low for SLM grammars and often the optimum value is around 9 or higher. Lower values are used when the SLM is not well trained.

2 Set `rec.Pruning` appropriately.

This parameter is highly correlated with the `rec.GrammarWeight` parameter. As `rec.GrammarWeight` increases, the recognizer runs faster and, typically, the pruning value needs to be increased to avoid extensive pruning.

3 Set `rec.pass1.gp.WTW` to `0` or `-50`.

This parameter is used to penalize word transition. It penalizes longer sentences in word loop type grammars—such as +[word1 word2 ...]).

In SLM-based grammars, the model probabilities already perform this task. Therefore, typically, this parameter should be set to `0` or `-50` instead of its default value of `-200`.

4 Set `rec.PPR` to `TRUE`, except for small vocabulary cases, such as name spelling.

5 If you observe a slow response time with large SLM grammars, try the following:

- Use the *English.America.1.3* master package.
- Turn N-best processing off (set `rec.DoNBest=FALSE`) or constrain the size of the N-best list (using `rec.NumNBest`) as much as possible.

Note: To get an N-best list with an SLM grammar, you must set `rec.Interpret=FALSE`.

The best way to determine the exact values of the parameters is to run multiple *batchrec* experiments with different settings to determine which values give the best performance. Keep in mind that large vocabulary SLMs typically run slower with extended master packages (packages with the suffix *.2*) than with standard master packages (packages with the suffix *.1*).

Measure the perplexity of a model

Perplexity is a measure of the quality of a language model. You can use this measure to tune the recognition accuracy of a model. The lower the perplexity, the better the model. In general, perplexity is correlated with recognition accuracy and a relative perplexity improvement (decrease) of 10% or more is indicative of better recognition accuracy.

Perplexity is interesting as a tuning tool because it is much faster to obtain than recognition accuracy, which requires running *batchrec*. To measure perplexity, you need a test set, which is a set of transcribed sentences from the application. This set must contain sentences that were not used to train the SLM.

To measure the perplexity of an SLM, use the *process-slm* tool, as follows:

```
process-slm
   slm_basename1 slm_basename2 ... slm_basenameN
   [-ppl-corpus test-set-filename]
```

Where:

- *slm_basename1 slm_basename2 ... slm_basenameN* are the filenames of the SLMs to measure. These are the *.slm* files obtained from *train_slm* without the *.slm* extension.
- *test-set-filename* is the name of the test set.

For example, suppose that two models were trained, a bigram model and a trigram model. You want to determine if the speed hit incurred with the trigram model is worth the performance gain. You can measure the perplexity of both models (*2gram.slm* and *3gram.slm*) on a test set called *test.txt* using the *process-slm* tool, as follows:

```
> process-slm 2gram 3gram -ppl-corpus test.txt
```

The output of the tool looks like:

```
Loading SLM file 2gram.slm
Done loading SLM file
Computing perplexity of corpus test.txt
```

```
8306 observation(s) logprob = -10285.300236 ppl = 17.310023
Perplexity = 17.310022
Loading SLM file 3gram.slm
Done loading SLM file
Computing perplexity of corpus test.txt
8306 observation(s) logprob = -10249.900334 ppl = 17.140981
Perplexity = 17.140982
```

In this case, the perplexity improvement going from a bigram to a trigram model is minimal: from 17.31 to 17.14. Since this is smaller than 10%, the trigram will probably not yield a significant recognition accuracy improvement.

Summary of the procedure for creating an SLM grammar

In summary, to create an SLM grammar:

1 Create a training set to train the language model. This text file, containing one phrase per line, should be representative of what a user of the target application would say. To create a class-based SLM:

a. Look at your training set to determine the appropriate class rules.

b. Write a GSL grammar that defines each of these rules. This grammar, called the *tagging grammar*, will be used to relabel the training set.

c. Create a tagging package using *nl-compile*. For example:

```
> nuance-compile flightinfo.tagging.grammar English.America
```

d. Run the *nl-tag-tool* utility to relabel the training set. For example:

```
% cat training.txt | nl-tag-tool -package flightinfo.tagging
-grammar .Tagging -no_output > training.relabeled.txt
```

2 Create a vocabulary for the model (optional). This vocabulary is used by *train-slm* to constrain the words incorporated in the model. The vocabulary should be a subset of the words used in the training set. If one is not supplied, *train-slm* will use all distinct words in the training set as the vocabulary.

3 Decide whether to use an open or a closed vocabulary.

4 Determine the order of the model.

5 Run *train-slm* to produce the statistical model in *.pfsg* and *.slm* formats. For example:

```
> train-slm -corpus training.relabeled.txt -vocab vocab -slm
flightinfo
```

6 Include the produced *.pfsg* file in a GSL grammar file.

7 Compile the GSL grammar using *nuance-compile*. For example:

```
> nuance-compile flightinfo.grammar English.America.1
-auto_pron -dont_flatten
```

8 Set appropriate parameters.

9 Measure the perplexity of the SLM using *process-slm*. For example:

```
> process-slm 2gram 3gram -ppl-corpus test.txt
```

You may need to repeat the preceding steps several times, until your speed and accuracy requirements are met.

SLM and dynamic grammars

SLM grammars can be used with dynamic grammars just like conventional GSL grammars. The only restriction is that the GSL string to be compiled and used in a dynamic grammar cannot start with the *.pfsg* file name.

For example, the following GSL string is not valid for dynamic compilation:

```
"appslm.pfsg"
```

but the following format allows it to be compiled dynamically:

```
SlmRule
SlmRule "appslm.pfsg"
```

For information about dynamic grammars, see "Dynamic grammars: Just-in-time grammars and external rule references" on page 25.

SLMs and just-in-time grammars

You can use an SLM in a just-in-time grammar. To do so, you include the *.pfsg* grammar in a static grammar, and then compile that grammar using the *nuance-compile-ngo* utility.

Robust natural language interpretation

SLMs are useful for recognizing free-style speech. But to understand the meaning of the spoken phrase, you still need to write grammars that fill slots with the appropriate values.

With the conventional NL parsing, writing these grammar rules can be a tedious task and defeats the advantages of using an SLM in the first place. To address this problem, Nuance offers a robust NL parsing capability that lets you write slot-filling grammars for the meaningful phrases *only*. The NL interpretation engine spots these meaningful phrases in the text output of the recognizer and fills the appropriate slots.

Coupling the robust NL interpretation with the SLMs gives users more flexibility in what they can say and improves the performance of systems with high out-of-grammar rates. Specifically, this technology allows the NL engine to:

- Interpret more spontaneous speech effects such as hesitations, dysfluencies, and out-of-grammar sentences
- Maintain a high level of accuracy while using SLMs
- Handle flexible responses typical of mixed-initiative systems

Full and robust interpretation

The conventional NL interpretation is obtained as follows:

- Audio data is collected by an application
- The audio data is passed to the recognition engine
- The recognition engine generates text out of the audio data
- The text is passed to the NL engine
- The NL engine produces an interpretation result

In this conventional operating mode, called the *full* mode, the recognition engine and the NL engine are driven by the same GSL, which defines the valid phrases and how the slots are filled. However, you can use two different grammars: one driving the recognition phase and another driving the interpretation phase.

For example, the recognition can use an SLM grammar, allowing the application to recognize a large range of user's speech. The text output by the recognition engine is then processed by the NL engine running a GSL grammar that parses certain phrases only—the meaningful ones—and fills the appropriate slots.

While the conventional NL parser requires a *full parse* of the speaker's sentence by a top-level grammar rule—all the words in the sentence must be matched by a single grammar rule—the robust parser eliminates this requirement. If a full parse is not found, the robust NL parser attempts to fill as many slots as it can from partial parses using subsentence phrase fragments.

For example, consider the grammar for the flight reservation application above and the following user's sentence:

"I'd like to um I want to go to boston tomorrow."

The speech recognition engine, driven by the SLM, recognizes the sentence and sends the result to the NL engine, which tries to interpret the text.

In full mode, the text is not parsed by the NL engine since the sentence is not completely parsed by the grammar, and therefore no slots are filled. The sentence will be rejected. However, in robust mode, the *destination* and *day* slots will be filled by “boston” and “tomorrow” respectively.

Using robust interpretation

To use robust interpretation, you need to:

- Write the recognition and interpretation grammars
- Enable the robust interpretation mode

These tasks are described below.

Writing the recognition and interpretation grammars

The first step is to define both the recognition grammar and the interpretation grammar. To assign the interpretation grammar to the recognition grammar, you link them using the character = (equal) as follows:

`Recognition_Grammar` = *`Interpretation_Grammar`*

For example, let’s extend the SLM-based travel application trained earlier with robust interpretation rules:

```
SLMRule:slm "flightinfo.pfsg" = InterpretationSubGrammar

InterpretationSubGrammar
      [City:s {<destination $s>}
      DayOfWeek [ sunday monday tuesday wednesday
               thursday friday saturday ] {<day $string>}
      Month [ january
            february
            march
            april
            may
            june
            july
            august
            september
            october
            november
            december] {<month $string>}]
```

This grammar specifies that:

- The SLM in the file *flightinfo.pfsg* is used by the speech recognition engine to generate a text result.
- The subgrammar *InterpretationSubGrammar* is used by the NL engine to fill the *destination*, *day*, and *month* slots based on the recognition output.

Note: The use of the = (equal) character in GSL grammars is not restricted to SLMs. The following example is also valid:

```
SubGrammar +[word1 word2 ... wordN] = [   today {<day today>}
                                           tomorrow {<day tomorrow>}
                                           ...]
```

Guidelines for writing grammar rules for robust interpretation

To take advantage of the robust parser mode, you must follow some guidelines when writing grammar rules.

The robust parser returns slots filled by grammar rules that parse phrase fragments. Therefore, to take advantage of robust parsing, Nuance recommends that you write grammar rules that fill slots with minimally valid phrase fragments.

The following example illustrates this recommendation. Consider the following GSL grammar:

```
.Sentence Date
Date:slm "date.pfsg" = (Month ?the DayOrd)
Month [january february march ... december] {<month $string>}
DayOrd [first {<ord 1>} second {<ord 2>}]
```

and the sentence:

“january uh first”

Here the sentence contains the filled pause “uh,” so in full operating mode the `Date` rule will not match that sentence, because the whole sentence was not matched.

However, the robust parser is able to match january using the `Month` rule—and therefore fills the slot *month* with the value january—and to match “first” using the *DayOrd* rule—and therefore fills the slot *ord* with the value “1.” This shows how robust parsing allows an application to recover from a filled pause and still return the correct NL interpretation.

Now consider what happens if the grammar is written this way:

```
Date:slm "date.pfsg" = (Month:m ?the DayOrd:d) {<month $m><ord
$d>}
Month [january february march ... december] {return($string)}
DayOrd [first {return(1)} second {return(2)}]
```

In this case, with the same sentence, no NL interpretations are returned because the *Date* rule does not match the sentence and that is the *only* rule that fills slots.

By writing the grammar in this way, you are stating that the *Date* rule is the minimally valid—that is, shortest—phrase fragment that is acceptable for filling the slots. To be valid, the whole sentence must be matched. So, the rule of thumb for using the robust parsing is to fill at the smallest possible subgrammar.

Note: As the application developer, you must make sure that the words used by the interpretation grammar are included in the vocabulary of the SLM (recognition) grammar. The *nuance-compile* tool does not check for this type of error, which can cause a significant impact on the performance of the application. For example, consider the grammar `InterpretationSubGrammar` on page 90. You must make sure that sure that words in the *City*, *DayOfWeek*, and *Month* subgrammars are included in the *flightinfo.pfs*g grammars, otherwise some of the words that fill the slots may never appear in the recognition result, therefore, the associated slot would never get the correct value.

Enabling robust interpretation

The next step is to direct the NL engine to spot the key phrases anywhere in the sentence by setting its interpretation mode to `robust` with the parameter `rec.InterpretationEngine`. This parameter can be set at runtime. Its default value is `full`, which directs the NL engine to use the normal interpretation mode.

Typically, you write your application so that it sets the value of `rec.InterpretationEngine` to `robust` when robust interpretation is to be performed and resets it back to `full` to resume the normal operation mode.

If a complete parse of a sentence is found, the robust NL will find the same result as the conventional full NL, but it may require extra computations. Nuance recommends you switch the `rec.InterpretationEngine` to `full` when the robust functionality is no longer required.

Ambiguity and robust interpretation

If you allow your application to handle free-style speech, it is likely that a given sentence will lead to multiple interpretations.

Consider, for example, the following grammar:

```
.Date [ (Month DayOrd)
(DayOrd ?of Month)
]
Month [ january~0.2 {<month jan>}
february~0.1 {<month feb>}
]
DayOrd [ first {<day 1>}
(twenty first) {<day 21>}
]
```

And consider the following sentence:

"uh january twenty first of february"

No grammar rule parses this ambiguous sentence completely, so the robust parser finds multiple partial interpretations. The following interpretations are produced (higher ranked first):

- `{<day 21> <month feb>}`—filled by the phrase "twenty first of february"
- `{<day 21> <month jan>}`—filled by the phrase "january twenty first"
- `{<day 1> <month feb>}`—filled by the phrase "first of february"
- `{<day 1> <month jan>}`—filled by the phrases "january" and "first"

The first thing to notice is that the robust parser never fills a slot more than once and always uses non-overlapping phrases. For example, the phrases "january twenty first" and "first of february" cannot be used in the same interpretation, because they share the word "first" and they would fill the *month* slot with the conflicting values of `jan` and `feb`.

Note: The number of the interpretations returned by the NL engine is controlled by the runtime settable parameter `rec.MaxNumInterpretations`. By default, up to 10 interpretations can be returned.

To understand the order of interpretations, we need to know that the robust NL engine ranks interpretations according to the following rules:

1 Maximize the number of words covered in a sentence.

2 Minimize the number of grammar rules used.

3 Maximize the probability of the phrase fragments (using grammar weights).

Therefore, interpretations are first ordered by the number of sentence words covered by a phrase, then by the number of rules used, and finally by the probability of the phrases used in the parse.

In the example above, the first interpretation is filled by the phrase "twenty first of february" since it covers four words. The second and third interpretations, corresponding to "january twenty first" and "first of february", cover three words. Note that "january twenty first" is ranked before "first of february" since, while they both cover three words with the single grammar rule *.Date* the grammar weight on "january" (0.2) is larger than that for "february" (0.1).

Finally, the last interpretation corresponds to the disjoint phrases "january" and "first". This interpretation has the lowest rank since it only covers two words and uses two grammar rules (*Month* and *DayOrd*) to do so.

Statistical NL technology

Although the combination of the SLM and robust NL technologies offer a very flexible tool to develop free-style dialog applications, you still need to write grammar rules for the robust NL engine to interpret the spoken utterance. Nuance has developed a statistical NL technology that can eliminate the need to write NL grammar rules for call routing applications where a caller is routed to the correct destination based on the spoken utterance.

In these applications, there is only a single slot to be filled and that slot usually corresponds to the topic or the correct destination. To take advantage of this technology, a large number of example utterances must be collected and then transcribed and tagged with the correct destination route. Nuance provides an offline training tool that operates on the training data. Through statistical methods, this tool discovers the correlations between routing destinations and key phrases. The output of the training tool is called a *router package*. To take advantage of the router package, the parameter `rec.RouterDirectory` is set to point to the router package when the server is started. At runtime, the application can choose to use the router to interpret the sentence by setting the value of the `rec.InterpretationEngine` to `router`.

If you are interested in implementing applications based on this statistical NL technology, please contact Nuance professional services.

Chapter 6

Compiling grammars

6

This chapter describes how to compile a grammar. It first presents the different ways of compiling a grammar. It then describes the master packages used in the process and other issues related to the generation of a recognition package.

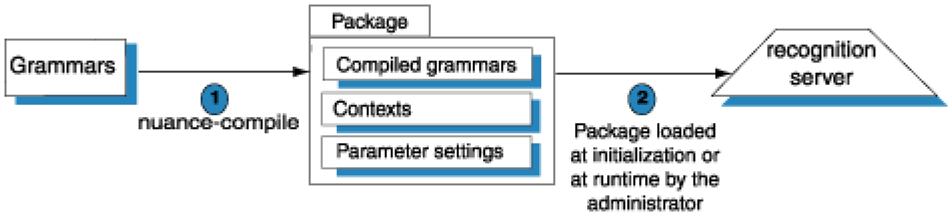
Compiling a grammar

Nuance offers five ways to compile and store a grammar:

- Static compilation (*nuance-compile*)
- Dynamic compilation to a database
- Dynamic compilation to a file (*nuance-compile-ngo*)
- Just-in-time compilation
- Implicit compilation

Static compilation

Static compilation is performed using the utility *nuance-compile*. This utility takes a grammar file, a master package, and other resources and produces a recognition package containing compiled grammars, contexts, and parameter settings. This package is loaded when a recognition server is started. This process is shown below.



To statically compile a grammar, enter the following command:

```
> nuance-compile package_name master_package [OPTIONS]
```

Where

- `package_name` specifies the path and root name for the package files to compile. It can be a relative or an absolute path. The *nuance-compile* tool compiles the file *package_name.grammar* and reads additional pronunciations from *package_name.dictionary* if that file exists. Nuance's natural language understanding capability requires the file *package_name.slot_definitions* and optionally other files. The package directory created by *nuance-compile* is named `package_name` unless the *-o* option is used.
- `master_package` is the name of a master package. Master package are provided by Nuance and found in *%NUANCE%\data\lang*. See "Choosing a master package" on page 103 for complete information on master packages.
- `[OPTIONS]` is any of the options described in "Summary of compiler options" on page 97.

For example, the following command tells the compiler to look for the grammar file *myapp.grammar* in the directory *c:\home\MyApp\grammars* and compile it using the North American English master package:

```
> nuance-compile c:\home\MyApp\grammars\myapp English.America
```

The grammar file passed to *nuance-compile* must contain at least one top-level grammar once all the `#include` directives have been expanded.

The package directory created by *nuance-compile* contains all the files that the Nuance System needs for speech recognition. The actual contents of this directory vary depending on the options that you have specified. The file *out.compile* is always created and contains the command line used to generate the package.

Listing available options

Executing *nuance-compile* without arguments returns information about its usage and default settings. Further usage information can be obtained by specifying the command with *-options*.

Specifying a name for the recognition package

By default, *nuance-compile* generates a package directory named `package_name` in the current directory. To specify a package name different from the default one, use the *nuance-compile* option *-o output_package_name*.

Summary of compiler options

The table below introduces some of the options to *nuance-compile*. For a complete list of options, see the *Nuance API Reference*.

Table 7: *nuance-compile* options

Option	Notes
-merge_dictionary `filename`	Merges standard dictionary entries with custom dictionary entries.
-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*.
-enable_jit	Enables the grammar for just-in-time compilation. If you do not specify this option, your application can perform recognition using the package's static grammars and contexts only.
-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.

Table 7: *nuance-compile* options

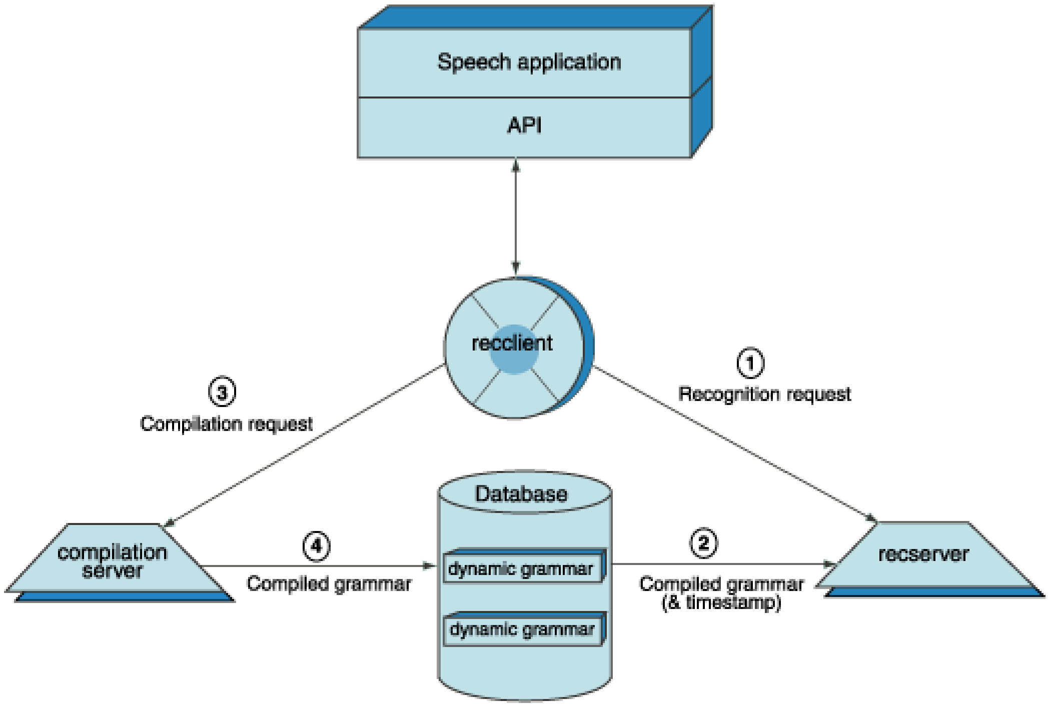
Option	Notes
-options	Displays all compiler options available. Use with no other option.

Dynamic compilation in a database

Dynamic compilation in a database is initiated programmatically with a call to the Compilation Server method `NewDynamicGrammar`. The grammar passed to this function is compiled into a database that the recognition server accesses as needed.

For more details on dynamic grammars, see "Creating a dynamic grammar in a database" on page 25.

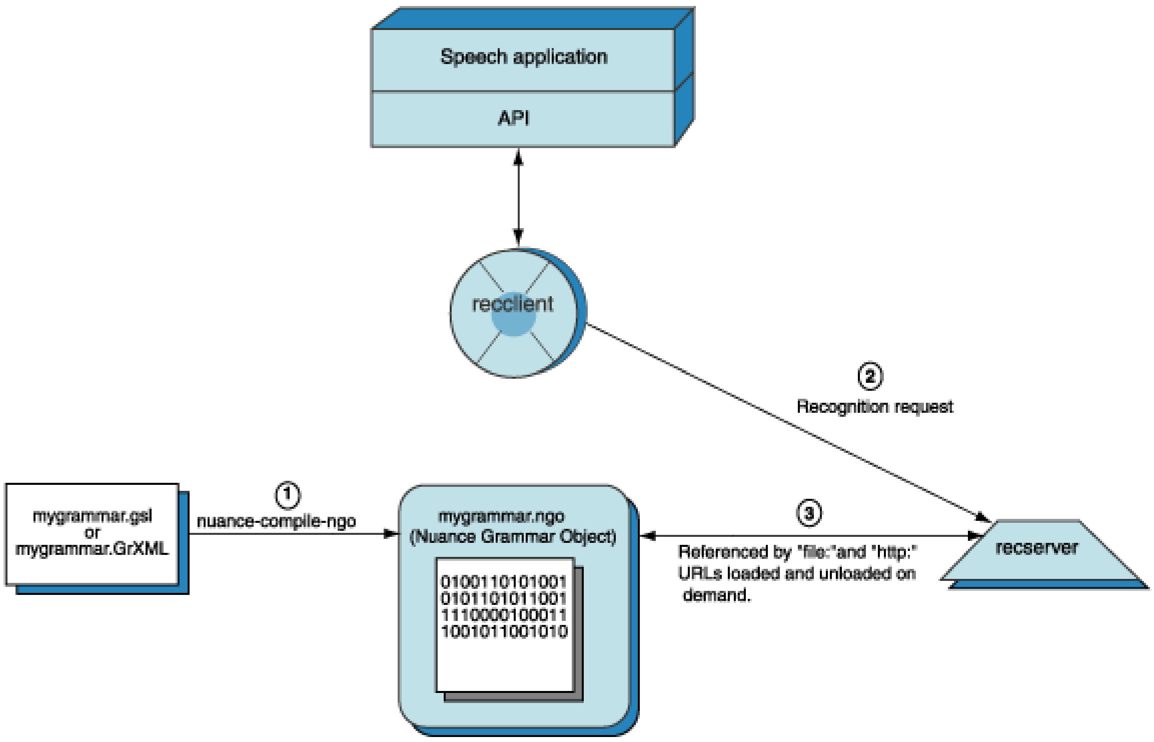
This process is shown below.



Dynamic compilation to a file

You can also compile a dynamic grammar with the utility *nuance-compile-ngo*. This utility can compile a GSL, GrXML (W3C format), or SLM grammar file into a *Nuance Grammar Object*—a binary grammar file with the *.ngo* extension. References to NGO grammars are done using the `file:` or `http:` protocols and are resolved by the recognition server, which loads and unloads these grammars on demand. Since an NGO file is precompiled, it is faster to load at runtime.

This process is shown below.


Speech application
API
recclient
2
Recognition request
mygrammar.gsl or mygrammar.GrXML
1
nuance-compile-ngo
mygrammar.ngo (Nuance Grammar Object)
0100110101001
0101101011001
1110000100011
1001011001010
3
Referenced by "file:"and "http:" URLs loaded and unloaded on demand.
recserver

nuance-compile-ngo

To compile a grammar with *nuance-compile-ngo*, enter the following command:

```
> nuance-compile-ngo gram_file_name -package pack_name [OPTIONS]
```

Where:

- *`gram_file_name`* is the name of the file containing the grammar to compile. The extension of this file should match the type of grammar. For example, a GSL file should have the *.gsl* extension and a VoiceXML grammar file should have the *.grxml* extension. If another extension is specified, the header of the grammar is examined to determine its type. If that fails, the grammar is parsed as a GSL grammar. If that also fails, the grammar is not compiled.

  `gram_file_name` is used as the base name of the output Nuance Grammar Object (*.ngo*) file created.

- `pack_name` is the name of just-in-time-enabled package that has been produced with *nuance-compile*, using the following options:
    - The master package that you want your dynamic grammars to use
    - The *-enable_jit* option

  The grammar file that you pass to *nuance-compile* to produce the package `pack_name` is irrelevant.
- `[OPTIONS]` includes the following:
    - `-do_crossword`, to generate compound word pronunciations for all words in the grammar.
    - `-dont_add_pauses`, to specify not to add pauses after words in the compile grammar. By default, pauses are added.
    - `-dont_flatten`, to direct the compiler not to flatten the grammar during compilation. By default, grammars are flattened.
    - `-dont_optimize_graph`, to direct the compiler not to perform any optimizations. By default, no optimization is performed.
    - `-dont_use_grammar_probs`, to disregard any grammar probabilities specified in the grammar file. By default, probabilities are used.

Note that when referring to an NGO grammar in a JIT grammar, the NGO grammar and the JIT grammar must have been compiled using the same master package. Otherwise, the reference to the NGO grammar will be treated as a `fail:` reference.

Compilation of just-in-time grammars

The compilation of a just-in-time grammar is initiated when you pass GSL or GrXML content directly to the recognizer at runtime through an API call, as shown in the following example:

```
String gsl = "[hello (hi there)]";
jsc.playAndRecognize(gsl);
```

where `jsc` is a *nuance.core.sc.NuanceSpeechChannel*.

The recognition server then:

- Resolves all references found in the just-in-time grammar, according to the rules and conventions described in “Assembling grammars” on page 102
- Compiles the resulting grammar
- Uses the compiled binary for recognition and discards it

This section describes the compilation options available for just-in-time grammars.

To compile a recognition package to use with a just-in-time grammar, run *nuance-compile* with the *-enable_jit* option, as follows:

```
> nuance-compile my.grammar English.America.1 -enable_jit
```

In this example, the *English.America.1* package can be used to perform recognition of a just-in-time grammar. If you do not specify the *-enable_jit* option, your application can only perform recognition using the package's static grammars and contexts.

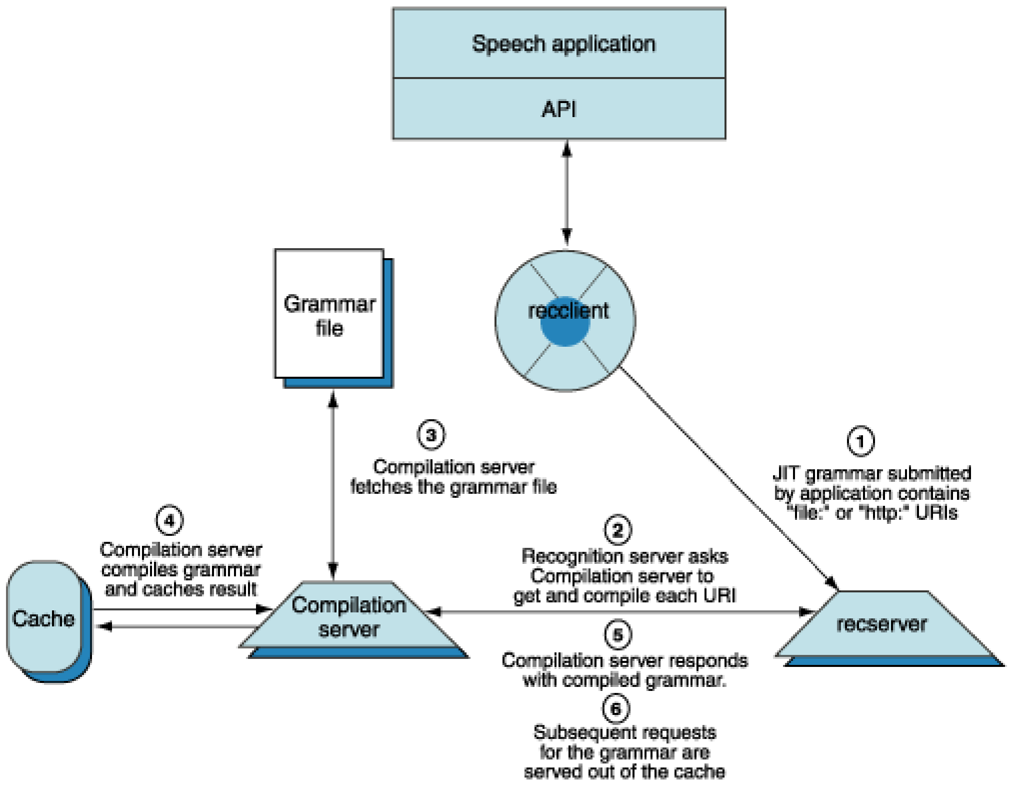
Implicit compilation

Implicit compilation is performed when a just-in-time grammar contains a `file:` or `http:` URI. Implicit compilation works as follows:

1. A just-in-time grammar submitted by the application contains a grammar reference in the form of a `file:` or `http:` URI.
2. The recognition server requests the compilation server to fetch and compile each of the references.
3. The compilation server gets the referenced grammars, compiles them, and caches the compilation result.
4. This result is passed to the recognition server, completing the request for compilation. Subsequent requests for grammars already compiled are served out of the cache.

The Recognition Server assembles the grammars that are used during recognition according to the rules and conventions described in "Assembling grammars" on page 102.

This process is below.


Speech application
API
Grammar file
recclient
1
JIT grammar submitted by application contains "file:" or "http:" URIs
2
Recognition server asks Compilation server to get and compile each URI
3
Compilation server fetches the grammar file
4
Compilation server compiles grammar and caches result
5
Compilation server responds with compiled grammar.
6
Subsequent requests for the grammar are served out of the cache
Cache
Compilation server
recserver

Assembling grammars

The recognition server resolves all references in a grammar specification. This process is called *assembling* a grammar.

The recognition server parses a grammar specified by an application and prepares to recognize with that grammar according to the following rules.

If the grammar has a name and it corresponds with a static grammar or context, the recognition server uses that grammar for recognition.

If the grammar is a just-in-time grammar, then the process is as follows:

- The recognition server requests the loading of all the components specified in references, until all references are resolved.
- The assembled grammar is compiled, used for recognition, and discarded after recognition ends, but the individual components are cached for later use.

- URI references specified with the `file:`, `http:`, and `dgdb:` protocols are always tested before each recognition.

Choosing a master package

The Nuance System provides multiple sets of acoustic-phonetic Hidden Markov Models also called *master packages*. These packages have been optimized for telephone-quality audio—that is, channels with 8-kHz/8-bit mulaw or alaw encoding, significant noise, narrow bandwidth (300 to 3300 Hz), and a variable linear frequency response. However, these packages also work well with most other microphones and audio channels.

Nuance provides master packages for multiple natural languages and packages that are optimized for use in several different types of environments. Master packages other than North American English are *not* included in the "Custom" Nuance install option. These packages need to be specifically downloaded from Nuance Technical Support Online (*support.nuance.com*).

Master package naming convention

The default name format for a given master package is:

language[.dialect][.package_type]

where the square brackets indicate an optional string. Examples of this format are:

- *English.UK*
- *German*
- *French.Canada.1*

Note: If a master package is for a language for which only a single dialect is supported or for which the dialect name is equivalent to the language name, the dialect modifier is omitted. For example, the master package supporting Canadian French speakers is *French.Canada*, while the master package supporting European French speakers is simply *French*.

The specific package name format is:

language[.dialect].package_type.version.release

All languages have master packages with a type number equal to 1, and this version is called the *compact* package. A language may have master packages with types 1, 2 and/or 3:

- Packages with type number 2 are called *extended* packages. Extended packages perform additional recognition processing that may improve accuracy but requires additional memory.
- Packages with type number 3 are called *enhanced wireless* packages. They are designed for use in all environments including hands-free environments, on cellular phones, and in other environments with high levels of background noises. Enhanced wireless packages are capable of robust handling of any type of incoming speech, including normal land lines, cellular phones, and hands-free equipment. They require additional memory over the compact and extended packages.

The *version* number is the version number of the package; this number increases when there are significant changes to the package, for example when new acoustic models are available. The *release* number is the release number of the package; this number increases when there are minor changes to the package, for example minor changes to the dictionary.

Typically, you should leave out the *version* and *release* numbers when referencing a master package—the compiler defaults to using the latest version and release. If you need to use a specific version and release of a master package, you can specify them explicitly.

To determine how default names are mapped to specific master package names, use the utility *nuance-master-packages*.

Language-specific packages

A master package is language specific. The Nuance System provides multiple packages for some languages and dialects, such as North American English. The package you use has an impact on your application's performance.

Language-specific packages provided with the current release include the following:

- American Spanish
- Australian/New Zealand English
- Brazilian Portuguese
- Canadian French
- Cantonese
- Catalan
- Czech
- Hebrew
- Italian
- Japanese
- Korean
- Mandarin (mainland China)
- Mandarin (Taiwan)
- North American English

- Danish
- Dutch
- European French
- European Spanish
- German
- German (Switzerland)
- Greek
- Norwegian
- Singapore English
- South African English
- Swedish
- Turkish
- U.K. English

Nuance releases new language-specific master packages as they become available. For a complete list of the master packages and languages supported, visit Nuance Technical Support Online (*support.nuance.com*).

Grammars for each language must be written in a specific locale, defined in the documentation provided with each language. The following languages must be run under a specific locale:

- Hebrew: In Windows, Hebrew_Israel.1255; in Solaris, “he”
- Korean: In Windows, Korean or Korean_Korea.949; in Solaris, “ko”

Recommended usage

The command-line program *nuance-master-packages* lists the master packages installed on your system and the default package names mappings.

To determine which master package to use, keep in mind the following guidelines:

- When present, enhanced wireless master packages are generally the default package for a language. These master packages provide robust, accurate recognition for all applications. For example, *English.America* and *English.America.3* both map to the latest version of the *English.America.3* master package.
- If your system has memory constraints, try using the compact package, for example *English.America.1*, which uses less memory than *English.America.3.*

Mapping special characters

You may specify how certain characters in the English character set map to characters in the target language character set. Specifically, the characters that can be mapped are (space-separated listing):

```
( ) < > = + - * / , . ! ? # ^ ; : { } $ “ ‘ & @ | _ %
```

This specification is written in a plain text file, one line per mapping, according to the following syntax:

- The first (ASCII) character in a line is one of the listed above.
- The remaining non-blank characters, possibly multiple-byte characters, define what the first character is mapped to. The map may be one-to-many, in which case the list of equivalents, in the target language, are space-separated.
- If the first and the second characters are the same, the line is considered a comment, unless it is a special compiler directive (see paragraph at the end of this section).

This file should be installed in the directory *%NUANCE%\data\locale*. The name of this file follows the syntax that specifies the locale in your particular system. For example, the name of the French locale is French-France@1252 on Windows and fr_FR@ISO-8859-1 on Unix systems. Accordingly, the mapping file can be named:

- *French*, *French-France*, or *French-France@1252*, on the Windows platform.
- *fr*, *fr_FR*, or *fr_FR@1252*, on Unix systems.

For example, the locale equivalent mapping file *fr_FR*, may contain the following lines:

```
"«»
;; make «guillemots» equivalent to double quotes
```

The special comment-like line:

```
;;fast
```

is not treated as a comment but directs the compiler to speed up the processing of characters *only* when the locale name contains the suffix ISO-8859-1 (so-called Latin-1 languages) on Unix systems or the suffix 1252 on Windows platforms. Specifying such a directive in a mapping file on systems where the locale deviates from that suffix requirement leads to compilation errors.

Support for multiple languages

The Nuance System supports multilingual recognition. This feature lets you build applications that allow speakers to:

- Select their language of choice at the first prompt
- Use the correct pronunciation of foreign words in multi-lingual regions, for example, using the French pronunciation of French names within an English utterance

- Pronounce foreign words in international applications, for example, letting English or German-speaking callers use a Japanese dialing application and have names correctly recognized with the caller's native accent

To implement an application that uses multiple languages, you must use a master package that includes acoustic models for all the languages you want to support. Nuance can generate special mixed-language master packages on request. If you are interested in this feature, contact Nuance Technical Support with details about the languages you want to use, the languages you want automatic pronunciation for, the Nuance version and the operating system you are using, and the type of mixed-language support your application requires.

Grammar

Note that languages can be combined in the same grammar without special markings. For example:

```
Welcome ( *@hes@   [ hello bonjour ] )
```

Adding missing pronunciations

The *nuance-compile* program finds pronunciations for each word in your grammar using a set of standard dictionary files. However, you might need to create grammars containing words that do not have pronunciations in the standard dictionary—for example, unusual names or domain-specific names.

If *nuance-compile* finds any words in a grammar file that have no pronunciations in the standard dictionary, the program displays an error and creates a file in the current directory called *package_name.missing*. This file lists each word in the grammar with no known pronunciation.

You can provide pronunciations for these words in one of two ways:

1. By creating a custom dictionary for the package.
2. By using the *nuance-compile* automatic pronunciation generator option, *-auto_pron.*

Creating a dictionary file

To create a customized dictionary for a recognition package, create a file in the package directory called *package_name.dictionary*. The easiest way is to rename or copy the *package_name.missing* file generated by *nuance-compile* to *package_name.dictionary* and place it in the corresponding package directory. Then add a pronunciation for each word using the phonetic symbols listed in "The Nuance phoneme set" on page 183. Each line in the file should contain one pronunciation—consisting of the word followed by its phonetic sequence, for example:

```
telegraph t E l * g r a f
```

You can use the Nuance utility *pronounce* to see pronunciations for similar words. For example, if you want to create a pronunciation for the name "Vaughan" you could get the pronunciation for "gone" by running:

```
> pronounce English.America gone
```

The program outputs the pronunciation:

```
gone g O n
```

From this you can determine that the pronunciation for "Vaughan" is "`v O n`".

To provide multiple pronunciations for a word, include each pronunciation on a different line. For example:

```
process p r A s E s
process p r o s E s
```

Multilingual dictionaries

A multilingual dictionary contains pronunciations in all the languages indicated and includes entries from the unilingual dictionaries. Words are not marked by language, while phonemes are tagged with a language code.

To create an application-specific dictionary, list the words and pronunciations as specified above, but tag the pronunciations with a language. Use phonemes from the languages you expect the callers accents to be in. For example, a person speaking French with a native North American English accent requires that the French words in your application be pronounced using North American English phonemes. The following dictionary entries illustrate how "hello" and "bonjour" can be handled in both the North American English and French accents:

```
hello h.EA E.EA l.EA o.EA
hello E.FR l.FR O.FR
bonjour b.EA *.EA n.EA Z.EA U.EA r.EA
bonjour b.FR o~.FR Z.FR u.FR r.FR
```

Automatic pronunciation is supported in one language only.

Merging and overriding dictionaries

You can specify your custom dictionary to interact with the master dictionary in one of two ways:

- *Override*, where word pronunciations in your dictionary replace those of the standard dictionary
- *Merge*, where alternate word pronunciations are added to the standard dictionary

To create an application-specific dictionary, see “Creating application-specific dictionaries” on page 183.

Command line directives

The *nuance-compile* option *-override_dictionary* lets you specify word pronunciations that replace any pronunciations existing in the standard Nuance dictionary. The following command line demonstrates how to use this feature:

```
> nuance-compile myGrammar English.America
         -override_dictionary c:\tmp\myDictionary.dictionary
```

The *nuance-compile* option *-merge_dictionary* allows you to add alternate pronunciations for words that already have a pronunciation in the standard Nuance dictionary. The following command line demonstrates how to use this feature:

```
> nuance-compile myGrammar English.America
            -merge_dictionary myAltProns.dictionary
```

The file specification for both of these options may be an absolute or a relative file path name.

If your application uses grammars that are compiled dynamically at run-time, compiling your package using the *-override_dictionary* and *-merge_dictionary* options will cause the runtime compiled grammars to use that custom dictionary.

Note: Detecting words that aren’t in the master dictionary or in your custom dictionary helps you find possibly misspelled words in your grammars.

Automatic pronunciation generator

The automatic pronunciation generator is a compiler feature that tries to create pronunciations for missing words. The pronunciations that the compiler generates, if any, are written into a file called *package_name.autopron*, and automatically included in your compiled package. Then, if you want to improve your pronunciations, you can examine the generated pronunciations, accept or improve them, and include them in your dictionary. (If you do not include them, they are overwritten next time you compile your package.)

To use this feature, recompile your package using the option *-auto_pron* on the *nuance-compile* command line as illustrated in the command:

```
> nuance-compile myGrammar English.America -auto_pron
```

You control the name of the output file where the missing pronunciations are written by using the option *-write_auto_pron_output*.

The following command line illustrates the use of this option. Here the compiler writes the automatically generated pronunciations to the file *myMissingWords* in the current directory:

```
> nuance-compile myGrammar English.America
            -auto_pron
            -write_auto_pron_output myMissingWords
```

The file specification for the *-write_auto_pron_output* option can be an absolute or relative file path name. If no pronunciations are generated automatically, any pre-existing *.autopron* file is removed.

Note: Using the option *-write_auto_pron_output* without the *-auto_pron* option has no effect.

Phrase pronunciations

You can often improve recognition accuracy by identifying compound words in your grammars. Also referred to as *crossword* or *multiword* pronunciations, compound-word pronunciations let you explicitly identify phrases that users tend to co-articulate. For example, users often say "give me" as "gimme," or "I want to" as "I wanna."

The grammar compiler *nuance-compile* and *nuance-compile-ngo* can use compound-word pronunciations for these types of phrases when they are specified in the dictionary.

Using *nuance-compile-ngo* is particularly efficient for dynamic grammars since it lets you generate crossword pronunciations for a specific subgrammar only, instead of generating them for the top-level grammar in a package and its entire contents.

For example, consider an application that uses a large top-level grammar, with a lot of fillers and alternate paths, but you only need to generate crossword pronunciations for one of the subgrammars. You can compile this subgrammar as a separate grammar document with *nuance-compile-ngo* and refer to the resulting NGO at runtime. Generating crossword pronunciations using *nuance-compile-ngo* provides more control and flexibility over the grammars that are compiled.

Note: If you want to use multiword pronunciations in your package, you must not use the option *-dont_flatten*.

For example, with the following grammar and dictionary specifications, multiple pronunciations are included for the phrase "what is":

```
; grammar specification
Balance (what is my ?account balance
; dictionary specification
what         w ^ t
is           I z
(what is)    w ^ z
(what is)    w ^ t z
```

The last two lines in the previous dictionary specification provide compound-word pronunciations explicitly in a dictionary file. The format to use is similar to the single word pronunciation format, except that the compound word should be enclosed in parentheses:

`(<compound_word_sequence>) <phoneme_sequence>`

You can also use compound-word modeling to improve recognition accuracy for sequences of very short words (where each word contains one or two phonemes). Using compound words in these situations allows the Nuance System to assign the most detailed context-dependent acoustic models possible. This usually yields better performance than using a sequence of individual words, where the many word boundaries result in the assignment of less-detailed models. Some examples of this type of pronunciation are:

```
(i b m)            aj b i E m
(a t and t)        e t i * n t i
```

See “Tuning recognition performance” on page 113 for a description of the trade-off between accuracy and recognition performance, when a grammar is compiled with the crossword option.

Note: Previous versions of the Nuance system supported the ability to explicitly specify compound-word pronunciations by using “new” words, like “what_is”, in your grammar and explicitly adding pronunciations for the word “what_is” in the dictionary. This mechanism is still supported. The underscore mechanism, used to create a compound word out of a sequence of several words, does not permit pauses between the individual words in the sequence. Therefore, you should use the parenthesis mechanism when you want *both* compound *and* isolated pronunciations of compound words to be supported.

Optimizing recognition packages

This section presents ways in which you can optimize your recognition packages, including:

- Minimizing the size of your compiled package
- Speeding up the compilation process

- Improving recognition

Creating unflattened grammars

Large grammars—especially those that contain large subgrammars frequently used by other grammars—create packages that consume a large amount of memory. This is because, by default, grammars are fully expanded so that a subgrammar referenced multiple times in a grammar is included multiple times in the binary representation.

Packages compiled with the *-dont_flatten* option have each subgrammar included only once in the binary representation no matter how many times that subgrammar is referenced. This mechanism can cause a significant reduction in the size of the binary files for a package—however, it can also slightly slow recognition (on the order of 5%) for some applications.

Note: You must use the *-dont_flatten* option if any of the grammars in your application use recursion. On the other hand, this option disables the use of multiword pronunciations.

Speeding up compilation

By default, *nuance-compile* performs extensive graph optimization prior to the actual compilation. This results in an efficient, faster runtime package. However, it takes more time to complete the compilation process. To speed up the compilation, you can turn the optimization passes off by using the option *-dont_optimize_graph*.

Filler grammars

GSL gives you the ability to specify a subgrammar as *filler*. When you use filler grammars in your package, the recognition engine scores the confidence of a phrase recognition using only non-filler words.

This can sometimes give you more accurate results by preventing correct recognition of filler phrases from boosting the score of a recognition result into an acceptable range. For example, if the recognizer hypothesizes that the out-of-grammar phrase “I want to leave next month” is actually the in-grammar phrase “I want to travel by car,” that result might be scored highly enough to be accepted because of the match of the “I want to” portion. If this portion is disregarded, the score of the remaining words should be low enough so that the utterance is correctly rejected.

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

  Using the *nuance-compile* and *nuance-compile-ngo* 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 master packages
- 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 116.

Chapter 7

Testing grammars

7

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 first section presents an overview of various grammar tests. The second 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.

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*
- *generate-nlref*
- *nl-tool*
- *Xapp*
- *batchrec*

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

Specifying a grammar to the command-line tools

The command-line tools described in this section take as input a recognition package, specified with the *-package* option, and a grammar, specified with the *-grammar* option. The package specified depends on the type of grammar. For most of the command-line tools described in this chapter, you can specify the following types of grammars:

- A static grammar
- A local grammar file
- An external reference to a grammar
- A just-in-time grammar

Specifying a static grammar

You can specify a static grammar as the *-grammar* option. In this case, the recognition package must be the package into which the grammar specified was compiled. For example

```
> generate -package MyPackage -grammar .MyGrammar
```

Specifying a local grammar file

The simplest way to test a grammar is to specify a local grammar file. For example:

```
> generate -package MyPackage -grammar date.gsl
rm.Addresses=localhost -config_name en-US
```

Where:

`MyPackage`	Is a recognition package enabled for just-in-time grammars. See "Enabling compilation of just-in-time grammars" on page 46 for more information.
`date.gsl`	Is a GSL or GrXML source grammar file located in the directory where you started the utility. This grammar must start with the GSL or GrXML header.
`rm.Addresses=hostname`	Refers to a resource manager. This is required to resolve external rule references. You can omit this option if your grammar does not include external rule references.
`-config_name en-US`	Refers to the compilation server used to resolve the external rule references. You can omit this option if your grammar does not include external rule references.

Specifying a just-in-time request from the call logs

If you want to run tests using the same just-in-time grammars created by your application, you can get them from the call logs. See the *Nuance Application Developer's Guide* for information on the location and structure of call logs.

For each recognition request, this information is provided in the GRAMMAR field of the call log. For example, the following shows a just-in-time request created for the PizzaTalk sample application:

```
.....
    GRAMMAR                    = RECOGNIZE
Nuance-Grammar-Label:ConfirmSizeAndToppings
Nuance-Package-Name:en-US
Nuance-Config-Name:en-US
Content-Base:http://localhost:8090/PizzaTalk/AppServ/dialogs/pizz
a.vxml
Vendor-Specific-Parameters:rec.Pruning="1250";client.TooMuchSpeec
hTimeoutSecs="10.0";client.NoSpeechTimeoutSecs="8.0";rec.Confiden
ceRejectionThreshold="50"

;GSL2.0
_VWS_:public[
<http://localhost:8090/PizzaTalk/AppServ/grammars/Confirm.gsl#Con
firm> {<vws_id 0>}
<http://localhost:8090/PizzaTalk/AppServ/grammars/universals.gsl#
Help>~0.01 {<vws_id 1>}
<http://localhost:8090/PizzaTalk/AppServ/grammars/universals.gsl#
Repeat>~0.01 {<vws_id 2>}
<http://localhost:8090/PizzaTalk/AppServ/grammars/universals.gsl#
Goodbye>~0.01 {<vws_id 3>}
<http://localhost:8090/PizzaTalk/AppServ/grammars/universals.gsl#
Cancel>~0.01 {<vws_id 4>}
]
...
```

The just-in-time grammar starts with the keyword `RECOGNIZE`. Save the just-in-time request to a text file. For example, you could save the just-in-time request above to the *top.gsl* file, as follows:

```
top.gsl
------------------------------------------------------
RECOGNIZE
Nuance-Grammar-Label:ConfirmSizeAndToppings
Nuance-Package-Name:en-US
Nuance-Config-Name:en-US
Content-Base:http://localhost:8090/PizzaTalk/AppServ/dialogs/pizz
a.vxml
Vendor-Specific-Parameters:rec.Pruning="1250";client.TooMuchSpeec
hTimeoutSecs="10.0";client.NoSpeechTimeoutSecs="8.0";rec.Confiden
ceRejectionThreshold="50"
```

```
;GSL2.0
_VWS_:public[
<http://localhost:8090/PizzaTalk/AppServ/grammars/Confirm.gsl#Con
firm> {<vws_id 0>}
<http://localhost:8090/PizzaTalk/AppServ/grammars/universals.gsl#
Help>~0.01 {<vws_id 1>}
<http://localhost:8090/PizzaTalk/AppServ/grammars/universals.gsl#
Repeat>~0.01 {<vws_id 2>}
<http://localhost:8090/PizzaTalk/AppServ/grammars/universals.gsl#
Goodbye>~0.01 {<vws_id 3>}
<http://localhost:8090/PizzaTalk/AppServ/grammars/universals.gsl#
Cancel>~0.01 {<vws_id 4>}
]
```

Make sure that the just-in-time grammar is formatted properly. It must start with the keyword `RECOGNIZE`, followed by a set of `<key>:<value>` pairs, followed by a blank line, followed by a GSL or GrXML document:

```
RECOGNIZE\n
header1:value\n
header2:value\n
...
\n
(grammar document -- starting with GSL or XML header)
```

Where `\n` represents a newline.

You can then refer to this file as follows:

```
> generate -package MyPackage -grammar top.gsl
rm.Addresses=hostname -config_name en-US
```

Specifying an external reference to a grammar

You can specify a reference to a public grammar rule. For example:

```
> generate -package MyPackage -grammar “<file:gram.gsl>”
rm.Addresses=localhost -config_name en-US
```

The `rm.Addresses` and *-config_name* options are required to resolve external rule references. The recognition package specified should be enabled for just-in-time grammars.

Specifying a just-in-time grammar

You can specify a just-in-time grammar directly on the command line by specifying within quotes. For example:

```
> generate -package MyPackage -grammar "[red green blue]"
```

The recognition package specified should be enabled for just-in-time grammars.

parse-tool

This tool lets you test whether your grammar parses a given sentence correctly. It takes sentences from standard input and returns a value indicating whether the sentence was successfully parsed.

To run *parse-tool*, specify the package and the grammar you want to analyze. The grammar can be any of the grammars described in “Specifying a grammar to the command-line tools” on page 117. Use the option *-print_trees* to get a display of the grammar paths traversed to match the utterance. For example:

```
> parse-tool -package numbers -grammar .N0-99 -print-trees
```

produces the following output:

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

generate

This tool 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 a grammar that you want to analyze. The grammar can be any of the grammars described in “Specifying a grammar to the command-line tools” on page 117.

For example:

```
> generate -package MyPackage -grammar date.gsl
```

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 the 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.

If you are using a Say Anything grammar, note that *generate* only generates sentences using the interpretation portion of a grammar package, without using any grammars in the recognition portion of the package.

If you are using robust interpretation to parse just the meaningful phrase fragments, *generate* will only generate valid phrase fragments in that grammar, without any extra filler words that could be included in phrases parsed by an SLM grammar.

generate-nlref

This program generates an NL transcriptions file from a transcription file. The transcription file lists a set of recorded utterances and the text transcription for each. The format for the transcription file is the same as the one used with the *-transcriptions* option for *batchrec*. See "`-transcriptions transcriptions_file`" on page 133 for more information.

When used with *batchrec*, *generate-nlref* generates the `-nl_transcriptions` input file used by *batchrec* for natural language error rate scoring (see "`-nl_transcriptions nl_transcriptions_file`" on page 134).

To run *generate-nlref*, specify the package and the grammar used to generate the transcriptions. The grammar can be any of the grammars described in "Specifying a grammar to the command-line tools" on page 117. For example:

```
> generate-nlref -package MyPackage -refs input.txt -nlref
output.txt -grammar MyGrammar.gsl rm.Addresses=localhost
-config_name en-US
```

You can also use *generate-nlref* to easily identify out-of-grammar and ambiguous utterances. Note that *generate-nlref* handles out-of-grammar sentences (*-oog* option) differently depending on the parsing mode:

- In full parsing mode, a sentence is out-of-grammar when no full parse was found for it.
- The robust parser is designed to work even with sentences for which a full parse cannot be found. Therefore, in robust parsing mode, a sentence is considered out-of-grammar when no interpretations are generated for any fragment, that is, no slots could be filled.

Since the robust parser attempts to find the best interpretations from parts of a phrase, ambiguous interpretations are more likely to be generated. Be aware of this last point when passing the *-ambig* option to *generate-nlref*. To limit the number of interpretations generated, set property `rec.MaxNumInterpretations` to an acceptable value on the *generate-nlref* command line. For example, setting `rec.MaxNumInterpretations=1` will return the top interpretation only.

nl-tool

This tool takes sentences from standard input and outputs the interpretations for them. To run *nl-tool*, specify the package and the grammar you want to analyze. The grammar can be any of the grammars described in "Specifying a grammar to the command-line tools" on page 117. For example:

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

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

This utility works in both full and robust interpretation mode. By default *nl-tool* operates in full parser mode. To enable the robust parsing mode, use the *-robust* command-line option.

Since the robust parser attempts to find the best interpretations from parts of a phrase, ambiguous interpretations will most likely be generated. For example, consider the grammar:

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

and the utterance:

"january february"

This phrase would fail to be parsed in full parser mode. However, the robust parser generates two valid interpretations for it:

```
{<month jan>}
{<month feb>}
```

See "Robust natural language interpretation" on page 88 for more information on robust parsing.

Note: The parameter `rec.MaxNumInterpretations`, used by *nl-tool*, can be used to limit the number of interpretations generated.

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.

Note: *Xapp* does not support dynamic grammars and just-in-time grammars.

To run *Xapp*:

1 Start a *recserver* process 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 *recognition server* 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.

- Say 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.

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 8 for more information on *batchrec*.

Chapter 8

Testing recognition performance

8

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 may 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. 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; if you are using just-in-time grammars, make sure that this package is enabled as described in "Enabling compilation of just-in-time grammars" on page 46.
- A file containing a list of the files to be processed.
- The *-rcengine* option, if your grammars include external rule references.
- The `rm.Addresses` property, which specifies the location of the resource management service; this resource management service should point to a grammar compilation service and a recognition service. This is required if your grammars include external rule references or if you are using just-in-time grammars.

For example:

```
> batchrec -package package_name -testset testset_list -rcengine
rm.Addresses=localhost
```

batchrec also supports a number of additional arguments, described in "Additional options" on page 133. 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 option *-grammar*.

Specifying a just-in-time grammar in the test set file

To specify a just-in-time grammar in a test set file, the grammar that you pass to the `*Grammar` keyword must start with the just-in-time header; for example:

```
*Grammar "RECOGNIZE
Nuance-Package-Name: en-US
Nuance-Config-Name: en-US
```

The grammar that you pass to **Grammar* must be formatted as follows:

- It must be specified inside quotes.
- There must be an empty line between the just-in-time header and the `;GSL2.0` header.
- There must not be an extra space after the `RECOGNIZE` keyword.

Note: For more information about the RECOGNIZER header, see "Specifying just-in-time grammar options: Using the keyword RECOGNIZE" on page 42.

Typically, you create command blocks, listing the set of files to recognize with each grammar. For example:

```
;Grammar using external rule references
;--------------------------------------
*Grammar "RECOGNIZE
Nuance-Package-Name: en-US
Nuance-Config-Name: en-US

;GSL2.0
MAIN:public <http://grammar.example.com/data/Digits.gsl>"
C:\data\1234.wav
C:\data\3456.wav
C:\data\8765.wav

;Grammar using a reference to a static grammar
;---------------------------------------------
*Grammar "RECOGNIZE
Nuance-Package-Name: en-US
```

```
Nuance-Config-Name: en-US

;GSL2.0
MAIN:public <static:Digits>"
C:\data\1234.wav
C:\data\3456.wav
C:\data\8765.wav

;Grammar using a reference to an SLM
;-----------------------------------
*Grammar "RECOGNIZE
Nuance-Package-Name: en-US
Nuance-Config-Name: en-US

;GSL2.0
MAIN:public <file:slm.ngo>"
C:\data\test1.wav
C:\data\test2.wav

;Grammar using a reference to a GrXML grammar
;---------------------------------------------
*Grammar "RECOGNIZE
Nuance-Package-Name: en-US
Nuance-Config-Name: en-US

;GSL2.0
MAIN:public <http://grammar.example.com/data/base.grxml>"
C:\data\test3.wav
C:\data\test4.wav
```

You can also use the just-in-time recognition request with the grammars active at the time of recognition, as described in “Specifying a just-in-time request from the call logs” on page 118. Instead of saving the just-in-time request in a file, pass it to the **Grammar* keyword between quotes. For example:

```
*Grammar ’RECOGNIZE
Nuance-Grammar-Label:ConfirmSizeAndToppings
Nuance-Package-Name:en-US
Nuance-Config-Name:en-US
Content-Base:http://localhost:8090/PizzaTalk/servlet/AppServ/dial
ogs/pizza.vxml
Vendor-Specific-Parameters:rec.Pruning="1250";client.TooMuchSpeec
hTimeoutSecs="10.0";client.NoSpeechTimeoutSecs="8.0";rec.Confiden
ceRejectionThreshold="50"

;GSL2.0
_VWS_:public[
<http://localhost:8090/PizzaTalk/servlet/AppServ/grammars/Confirm
.gsl#Confirm> {<vws_id 0>}
<http://localhost:8090/PizzaTalk/servlet/AppServ/grammars/univers
als.gsl#Help>~0.01 {<vws_id 1>}
<http://localhost:8090/PizzaTalk/servlet/AppServ/grammars/univers
als.gsl#Repeat>~0.01 {<vws_id 2>}
<http://localhost:8090/PizzaTalk/servlet/AppServ/grammars/univers
als.gsl#Goodbye>~0.01 {<vws_id 3>}
<http://localhost:8090/PizzaTalk/servlet/AppServ/grammars/univers
als.gsl#Cancel>~0.01 {<vws_id 4>} ]'
```

Note: If the grammar text includes double quotes (as in the example above), use single quotes around the grammar passed to **Grammar*. For example:

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.

`*Interpret` *`text grammar`*

Interprets the text against the grammar specified. If the text is more than one word, include it in double quotes. If you are specifying a just-in-time grammar, put the grammar in single quotes.

`*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: Just-in-time grammars and external rule references" on page 25 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.

Note: A compilation server is required for the `*NewDynamicGrammar` functions. Make sure that you start *batchrec* with the `-rcengine` option (see "`-rcengine`" on page 135), that you start a resource manager and a compilation server, and that you set the rm.Addresses parameter correctly. See the *Nuance Application Developer's Guide* for more information.

`*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 grammar content. This can be a GSL extended right-hand side or a complete GSL or GrXML grammar document.

`*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 file and a natural language transcription file when you start *batchrec*, using the `-transcriptions` and `-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 “Additional options” on page 133 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.

See “Example: Using batchrec to perform enrollment” on page 139 for an example of performing enrollment with *batchrec*.

`*EnrollCommitPhrase`

Commits an updated dynamic grammar to the database.

`*EnrollAbortPhrase`

Ends an enrollment session without updating the dynamic grammar.

`*ModifyPhrase` *`db_handle db_key old_phrase_id new_phrase_id new_phrase_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`).

Additional options

batchrec supports a number of options:

`-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 54. 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*.

`-sv_output_scorefile filename`

When testing speaker verification functionality, specifies the output file to write verification data to. You use this output file to generate the final verification package. See the *Nuance Verifier Developer's Guide* for more information.

`-sv_transcriptions filename`

Lets you provide a file of transcriptions to use to score the accuracy of speaker verification operations. The transcription file lists one or more audio files (one per line) and the database key of the voice model of the speaker who created that file, for example:

```
/data/caller10/utt2.wav sv555221111.model
```

Any other models are considered impostors. See the *Nuance Verifier Developer's Guide* for more information.

`-sv_identification_transcriptions filename`

Lets you provide a file of transcriptions to use to score the accuracy of speaker identification operations. This file lists the contents of the identification groups. For example, the following lists a set of groups and their corresponding voiceprints:

```
Group_1 speaker_A.model speaker_B.model
Group_2 speaker_C.model speaker_D.model
```

You must use the *sv_identification_transcriptions* file with the *sv_transcriptions* file to indicate to which voiceprint an utterance belongs. See the *Nuance Verifier Developer's Guide* for more information.

`-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.

`-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.

`-detailed_nl_scoring`

Keeps track of the insertion, deletion, and substitution errors at the semantic level, when your application uses the robust NL parsing. For details on robust NL parsing, see "Robust natural language interpretation" on page 88.

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
         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.

Using batchrec for SLM grammars

The *batchrec* program determines the status of an utterance to be correct, incorrect, or reject by comparing NL interpretations to NL transcriptions.

Switching the recognition engine to the robust NL parsing mode may cause some NL results to be partially correct, in the sense that some slots contain the right values but others are either not filled or incorrectly filled.

Specify the *batchrec* command-line option *-detailed_nl_scoring* to keep track of the insertion, deletion, and substitution errors at the semantic level.

Example: Using batchrec to perform enrollment

To perform enrollment with *batchrec*, you need to create the following files:

- A testset file, which includes commands to enroll a set of utterances
- A transcription file, which lists all the utterances used for enrollment and associates a phrase ID with each utterance
- A natural language interpretation file, which specifies the natural language interpretation for each utterance

You then run *batchrec* with the following options:

```
> batchrec -package package_directory
        -testset testset_list_file
        -transcriptions transcriptions_file
        -nl_transcriptions nltranscriptions_file
```

For example, the following sample testset file opens a file system database and creates a record for a new speaker (`acct_1234`). It then performs enrollment of two difference phrases using the *.Enroll* enrollment grammar; three utterances are used to train each phrase. Finally, it closes the dynamic grammar database.

```
*OpenDynamicGrammarDatabase 1 -dbprovider fs -dbroot . -dbname
enroll_db -dbclass dgdb
*NewDynamicGrammarEmpty 1 acct_1234 1
*EnrollNewPhrase .Enroll 1 acct_1234 needed
utt01.wav
utt02.wav
utt03.wav
*EnrollCommitPhrase
*EnrollNewPhrase .Enroll 1 acct_1234 needed
utt04.wav
utt05.wav
utt06.wav
*EnrollCommitPhrase
*CloseDynamicGrammarDatabase 1
```

The phrase ID associated with each utterance is specified in the transcription file as follows:

```
utt01.wav word_1
utt02.wav word_1
utt03.wav word_1
utt04.wav word_2
utt05.wav word_2
utt06.wav word_2
```

The natural language interpretation for each phrase is specified in the natural language interpretation file as follows:

```
utt01.wav word_1 <name "word_1"> <phone_number "1234567">
utt02.wav word_1 <name "word_1"> <phone_number "1234567">
utt03.wav word_1 <name "word_1"> <phone_number "1234567">
utt04.wav word_2 <name "word_2"> <phone_number "7654321">
utt05.wav word_2 <name "word_2"> <phone_number "7654321">
utt06.wav word_2 <name "word_2"> <phone_number "7654321">
```

Chapter 9

Using the Analysis and Tuning Tools

9

Nuance provides the Analysis and Tuning Tools, a set of command-line tools to analyze the performance of an application and run tuning experiments. This chapter provides an overview of the tools and describes how to use them.

Some notes:

- The tuning tools are most appropriate for applications such as directed dialog and mixed-initiative applications. They are not recommended for verification, voice-enrolled recognition, and Say Anything voice-enrolled applications.
- You can also use batchrec to analyze the performance of an application if the tests you want to run are not covered by the tuning tools. See Chapter 8 for more information.

Overview

The Analysis and Tuning Tools—*nuance-analyze* and *nuance-tune*—let you analyze how your system is performing in deployment. By looking at different reports, you can get information on the number of calls made to the system, latency values, length of utterances, and so on, and then use this information to identify problem areas. The tools then allow you to run different recognition experiments—by trying out new grammars, parameters, recognition packages—that help you tune your system.

The tools take as input three types of data:

- Call log data—Call logs are text files that record all recognition activity performed during a single call session. See the *Nuance Application Developer's Guide* for information on turning on call logging.

- Transcriptions—A transcription is the written text corresponding to the phrase spoken by the caller. See Appendix E, "Nuance transcriptions conventions," for more information.
- Recognition package—The recognition package used in deployment or modified during the analysis and tuning process. See Chapter 6 for more information on recognition packages.

Based on this input, the tools generate fifteen different reports. There are two types of reports:

- **Unsupervised reports** are reports based on call log data and the recognition package, without transcriptions. These reports can provide insight into which parts of an application seem to be working well, as well as which parts might warrant deeper investigation.
- **Supervised reports** are based on call log data, transcriptions, and the recognition package. The tools load all available transcriptions and attempt to match them with the provided call log data. The supervised reports allow you to investigate in more details the problem areas identified with the unsupervised reports.

These tools generate reports as XML documents. The tools include default stylesheets that let you view the reports as HTML, simply by viewing the XML documents in Internet Explorer v6.0. You can also use tools such as Xalan or Saxon to convert the XML output to HTML, PDF, and RTF using the Extensible Stylesheet Language (XSL).

To configure the tools, you use a file called *tuner-config*. This file specifies:

- The package, call logs, and transcriptions to use
- General parameters used when processing and scoring the logs (for example, whether certain slots should be ignored during scoring)

Note: To view the XML reports as HTML, you must use Microsoft Internet Explorer 6.0; the tools rely on Internet Explorer's embedded XSLT engine. To check which version of Internet Explorer is installed, open Internet Explorer and click Help->About Internet Explorer. If you don't have Internet Explorer 6, you can also use an XSLT engine such as Saxon or Xalan to transform the reports offline. See "Processing XML reports" on page 175 for more information.

Using the tools: Getting started

This section helps you get started with the tools. It then provides references to detailed descriptions for each of these steps.

1 Run *nuance-analyze* on the command line, with no options:

```
> nuance-analyze
```

This command creates a sample *tuner-config* file that you can customize for your project.

2 Edit the sample *tuner-config* file as follows:

a Set parameter `Package` to point to your own recognition package.

b Set parameter `BaseLogDir` to point to your call logs.

c Set parameter `TranscriptionsFile` to point to your transcriptions file, if they are available. Otherwise, comment out this parameter.

d If you are using just-in-time grammars, edit the `CSArgs` parameter to point to the package specified in `Package`. If you are not using just-in-time grammars, comment out parameter `CSArgs`.

3 Run *nuance-analyze* again:

```
> nuance-analyze
```

4 In Internet Explorer 6.0, open the *index.html* file located in the output directory (*analysis/index.html*).

How to use this document

The remainder of this chapter provides more detailed information, as follows:

- "Preparing the input data" on page 144 describes how to make sure the transcriptions are in the correct format and how to filter your input data.
- "Editing the tuner-config file" on page 150 describes the parameters you can set in your configuration file.
- "Running nuance-analyze and nuance-tune" on page 157 describe the command-line options available for the tools.
- "Looking at reports" on page 161 describes the reports generated with the tools and explains how to process the XML output.
- "Working with the Records and Scores reports" on page 176 describes the format of the Records and Scores reports generated by the tools and explains how you can use them.

Preparing the input data

The tools use call log data and transcriptions as input to generate the reports. This section describes how to prepare the input data; in particular, it describes:

- How to make sure the transcriptions are in the correct format
- How to organize your data into groups
- How to filter call log data
- How to specify grammar overrides

Providing transcriptions

To investigate in more details the problem areas identified with the unsupervised reports, you can provide transcriptions. The transcriptions must follow the standard Nuance transcriptions conventions. See Appendix E, "Nuance transcriptions conventions," for more information.

Organizing the input data into groups

For most reports, the data is categorized by *group*—a way to group the utterances that correspond to the same recognition task. A group is usually closely associated with the grammar used to recognize the data.

You specify how the utterances will be grouped by setting the `GroupKey` parameter in the *tuner-config* file. By default, this is set to the value of the `GRAMMAR_LABEL` field in the call logs, which applies to just-in-time grammars. The `GRAMMAR_LABEL` field is obtained from the Nuance-Grammar-Label key in the just-in-time header.

If the `GRAMMAR_LABEL` field in the call logs is not set, the tools use the `GRAMMAR` field of the call record, which is set to the grammar used for recognition. This is appropriate for static grammars.

For Nuance 8.5, if you are using just-in-time grammars, you need to set the Nuance-Grammar-Label key in the just-in-time header to a meaningful value, for example the name of the current recognition state (e.g., GetPizzaSize). Otherwise, if there is no `GRAMMAR_LABEL` field, the tools will use the value of the `GRAMMAR` field in the call logs, which contains the entire just-in-time request for just-in-time grammars (and thus is not very useful).

You can also set the GroupKey to a different key. For example, standalone SpeechObjects applications typically use the `RULENAME` key.

Note: Integrators using a highly dynamic environment (for example, a VoiceXML interpreter) are strongly encouraged to populate the GRAMMAR_LABEL field or populate a similar field and specify it using the GroupKey parameter to help group records as appropriate.

Filtering call log data

By default, the tools process all the call log data located in the directory specified with *tuner-config* parameter BaseLogDir. However, it might be useful to process only a subset of this call log data. You can filter the data based on the date and group of the call, or by specifying a dataset file that lists the calls and utterances to process.

Filtering data based on the date and group

You can filter the data based on the date and group of the call by using the following command-line options when starting the tools:

- *-begin_date*
  The beginning date (inclusive) of the range of calls to process. Defaults to the first provided call. The format of this argument is "MON DD YYYY" (for example, "Jan 28 2003").
- *-end_date*
  The end date (inclusive) of the range of calls to process. Defaults to the last provided call. The format of this argument is "MON DD YYYY" (for example, "Jan 28 2003").
- *-group*
  The group(s) to process for analysis. You can specify this option multiple times to filter the groups being processed. If this option is not specified, all available groups are processed. See "Organizing the input data into groups" on page 144 for more information.

Make sure to specify the dates in double quotes (" ") for the string to be processed as a single argument.

Examples

```
> nuance-analyze -begin_date "Jan 28 2003" -end_date "Jan 30 2003"
-group Main -group Confirm
> nuance-tune -begin_date "Jan 28 2003" -group Main -group Confirm
-group Transfer
```

Filtering data based on a dataset file

You can also filter data by specifying a dataset file that lists the calls and utterances to process. You specify this file with the -dataset option.

The dataset is an XML file with a root <dataset> element, with <call> elements as its children. The <call> element can have the following attributes:

- `log`—Specifies the full path to the call log.
- `utts`—Specifies whether the tools should process all the utterances in the call (`utts="all"`) or only selected utterances (`utts="selected"`). If only selected utterances should be processes, these utterances are specified within `<utt>` children of the containing `<call>` element. In this case, the utterance filename should be specified in the `filename` attribute of `<utt>`.

  Note: You must specify the full path to the call log in the `log` attribute. However, when specifying the utterance filename in the `filename` attribute, you can specify a relative path, but the path *must* contain some of the date information (year, year/date, etc.).

The following example shows a sample dataset file:

Note: To look at the Document Type Definition (DTD) for the dataset file, see "DTD for the dataset file" on page 232.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<dataset>
   <call log="C:\Nuance\V8.5.0/tuner/samples/data/PizzaTalk/logs
      /Nuance/PizzaTalk/2003/02February/20/12/09-29-zamboni-001_5060-LOG"
       utts="all"/>
   <call log="C:\Nuance\V8.5.0/tuner/samples/data/PizzaTalk/logs
      /Nuance/PizzaTalk/2003/02February/20/12/11-50-zamboni-001_5060-LOG"
      utts="all"/>
   <call log="C:\Nuance\V8.5.0/tuner/samples/data/PizzaTalk/logs
      /Nuance/PizzaTalk/2003/02February/20/12/13-42-zamboni-001_5060-LOG"
      utts="all"/>
   <call log="C:\Nuance\V8.5.0/tuner/samples/data/PizzaTalk/logs
      /Nuance/PizzaTalk/2003/02February/20/12/15-05-zamboni-001_5060-LOG"
      utts="all"/>
</dataset>
```

Note: You can also create a call-based dataset from the Records or Scores report, using the *records_dataset.xsl* stylesheet. This stylesheet creates a dataset with all the records specified with the parameter `BaseLogDir`. You can then edit this dataset to specify only the records and utterances you want to process. For example, the dataset file above was created from the sample data provided with the tools.

This dataset is very useful since it allows you to divide your data into a training set, development set, and test set, and then perform experiments on each set. You can also use it to run a quick experiment on a small amount of data, for example, just the utterances for a name whose pronunciation you are tuning.

Specifying grammar overrides

By default, the tools look for the grammar specified in the call logs to process the input data. If you used just-in-time grammars, you might not have access to the original grammars used in deployment. For example, grammar *http://www.example.com/grammars/date.gsl* may no longer be available or its content may now be different from what it was in deployment.

If you have the sources for these grammars, you can still perform analysis and tuning by using the grammar override feature, which lets you specify a different grammar location in a *grammar overrides file*. This XML file can also include new parameter values to use when running recognition experiments with *nuance-tune*.

Creating the grammar overrides file

You create the grammar overrides file by using the *groups.xml* report and editing it as follows:

- Change the root element of the Groups report from `<report name="groups">` to `<grammar-overrides>`. Make sure to change the `</report>` end tag to `</grammar-overrides>`.
- Rewrite the grammars under the `<grammar>` elements as appropriate.
- To change dynamic grammar insertions, rewrite the `<insert>` elements under the `<dynagrams>` element within a grammar definition, as appropriate.
- To override parameters, edit/add/delete `<param>` elements under the `<parameters>` element. Note that you can only specify run-time settable parameters.
- To set parameters across all groups, specify `<param>` elements under the `<default-parameters>` section. Any parameter set in this section will be used across all groups unless that group overrides that parameter. Note that only one `<default-parameters>` section is allowed per grammar overrides.

  Again, the output Groups report will reflect the `<default-parameters>` specified in the grammar overrides. In addition, any default parameters set in the package's static context, if any, will also show up in the `<default-parameters>` section of the output Groups report.

You can use environment variables in your grammar definition, but not in your parameters definition. This is useful when you want to specify paths in external rule references.

To look at the Document Type Definition (DTD) for the grammar overrides file, see "DTD for the grammar overrides file" on page 227.

Priority of parameter settings

The following list shows the order of precedence (from highest to lowest precedence) used by the tools if a parameter is set in multiple places:

1 Parameters set in `<parameters>` elements for a grammar definition in the grammar overrides file.

2 Parameters in a named context in the package's static context (*<package>/nuance-resources*).

3 Parameters set in the `<default-parameters>` section in the grammar overrides file.

4 Default parameters set in the package's static context (*<package>/nuance-resource*).

Note: See the *Nuance Application Developer's Guide* for more information about setting parameters through contexts.

Example

The example below demonstrates how to create the grammar overrides file. The original *groups.xml* report produced by *nuance-analyze* for an application called PizzaTalk is simply copied to *pizza-grammars.xml* and the root element changed as appropriate:

File *groups.xml*:

```
<?xml version="1.0" encoding="US-ASCII"?>
<?xml-stylesheet href="groups.xsl" type="text/xsl"?>

<report name="groups">
  <group name="ConfirmDeliveryNow"
         num="4">
    <grammar-def instance="default">
         .....
```

File *pizza-grammars.xml*:

```
<?xml version="1.0" encoding="US-ASCII"?>
<grammar-overrides>
  <group name="ConfirmDeliveryNow"
         num="4">
    <grammar-def instance="default">
         .....
```

All of the grammars are then rewritten to point to local copies in *%NUANCE%\tuner\samples\data\PizzaTalk\grammars\*:

File *groups.xml*:

```
       <grammar>
        <![CDATA[RECOGNIZE
Nuance-Grammar-Label:ConfirmDeliveryNow
Nuance-Package-Name:en-US
Nuance-Config-Name:en-US
Content-Base:http://voicexml.nuance.com/samples/tuning/PizzaTalk/
grammars/
Vendor-Specific-Parameters:rec.Pruning="1250";client.TooMuchSpeechTimeout
Secs="10.0";client.NoSpeechTimeoutSecs="8.0";rec.ConfidenceRejectionThres
hold="50"

;GSL2.0
_VWS_:public[
<http://voicexml.nuance.com/samples/tuning/PizzaTalk/grammars/Confirm.gsl
#Confirm> {<vws_id 0>}
<http://voicexml.nuance.com/samples/tuning/PizzaTalk/grammars/universals.
gsl#Help>~0.01 {<vws_id 1>}
<http://voicexml.nuance.com/samples/tuning/PizzaTalk/grammars/universals.
gsl#Repeat>~0.01 {<vws_id 2>}
<http://voicexml.nuance.com/samples/tuning/PizzaTalk/grammars/universals.
gsl#Goodbye>~0.01 {<vws_id 3>}
<http://voicexml.nuance.com/samples/tuning/PizzaTalk/grammars/universals.
gsl#Cancel>~0.01 {<vws_id 4>}
]]]>
      </grammar>
```

File *pizza-grammars.xml*:

```
       <grammar>
<![CDATA[RECOGNIZE
Nuance-Grammar-Label:ConfirmDeliveryNow
Nuance-Package-Name:en-US
Nuance-Config-Name:en-US
Content-Base:file:/$NUANCE/tuner/samples/data/PizzaTalk/grammars/
Vendor-Specific-Parameters:rec.Pruning="1250";client.TooMuchSpeechTimeout
Secs="10.0";client.NoSpeechTimeoutSecs="8.0";rec.ConfidenceRejectionThres
hold="50"

;GSL2.0
_VWS_:public[
<Confirm.gsl#Confirm> {<vws_id 0>}
<universals.gsl#Help>~0.01 {<vws_id 1>}
<universals.gsl#Repeat>~0.01 {<vws_id 2>}
<universals.gsl#Goodbye>~0.01 {<vws_id 3>}
<universals.gsl#Cancel>~0.01 {<vws_id 4>}
]]]>
      </grammar>
```

In addition, parameters are tweaked for the ConfirmDeliveryNow group:

File *groups.xml*:

```
      <parameters>
        <param name="client.NoSpeechTimeoutSecs"
               value="8.0"/>
        <param name="client.TooMuchSpeechTimeoutSecs"
               value="10.0"/>
        <param name="rec.ConfidenceRejectionThreshold"
```

```
              value="50"/>
       <param name="rec.Pruning"
              value="1250"/>
     </parameters>
```

File *pizza-grammars.xml*:

```
     <parameters>
       <param name="rec.Pruning"
              value="1200"/>
     </parameters>
```

When a recognition experiment is run with this grammar overrides file, *nuance-tune* will use the grammars and parameters specified in this file (save for the “frozen parameters”, such as `rec.ConfidenceRejectionThreshold`; see “Setting recognition parameters for the recognition experiments” on page 159).

Grammar instances

Most applications have one unique grammar per group. However, if an application has more than one grammar in the same group—for example, different just-in-time requests or dynamic grammar insertions—you can override individual instances of these grammars within that group.

The Groups report, and therefore the grammar overrides file, would then contain one or more `<grammar-def>` elements; `<grammar>`, `<dynagrams>`, and `<parameters>` elements are then included in the `<grammar-def>` element. Each `<grammar-def>` is identified by an “instance” attribute; the tools compute a checksum of the grammar string (plus any dynamic grammar insertions) itself to produce this identification.

To override a specific instance of a grammar definition within a group, you can do so by editing the definition under the `<grammar-def>` matching that instance; you can also label one of the instances as the “default” instance if desired.

When applying grammar overrides, the tools compare the instance computed for the grammars seen in the call logs with the instances specified in the grammar overrides file. If a grammar seen in deployment matches one of the instances in the overrides file, the overridden definition is used in the experiment. If the grammar seen in deployment does not match any instances in the overrides file, but a “default” instance was provided, the default grammar definition is used. Otherwise, the grammar as seen in deployment is used as is.

Editing the *tuner-config* file

The *tuner-config* file provides configuration information used by the tools to determine:

- The input data—call logs and transcriptions files—to process for analysis and tuning
- Parameters for processing and scoring the data

When you start a tool, it looks for the configuration file specified with the command-line option *-config*. If this option isn't specified, the tool looks in the directory where it was started for a file called *tuner-config*. If it can't find it, it creates a new one by making a copy of the default configuration file—*%NUANCE%\tuner\tuner-config*—in the directory where you started the tool. This default configuration file points to the sample data provided with the tools.

At a minimum, you need to specify:

- The directory where the call log data is located
- The transcriptions files (optional for *nuance-analyze*)
- The location of the compiled package used to process the data

This section describes the format of the *tuner-config* file and the parameters that you can set. It also provides a sample *tuner-config* file.

Summary

The following table summarizes the parameters that you can specify in the *tuner-config* file.

Note: The Multiplicity column specifies whether a parameter can be set only once (Single) or many times (Multiple).

Parameter	Description	Type	Multiplicity	Required
`BaseLogDir`	Specifies the directory where the call log data is located.	STRING	Multiple	Required
`Transcriptions File`	Specifies a file containing transcriptions for the utterances to analyze.	STRING	Multiple	Optional in *nuance-analyze* Required in *nuance-tune*
`Package`	Specifies the location of the recognition package used to process the data.	STRING	Single	Required
`CSArgs`	Specifies how many compilation servers to start and what the command-line options should be for each.	STRING	Multiple	Optional

Parameter	Description	Type	Multiplicity	Required
`BuiltinContext`	Specifies the location of the built-in grammars in the system.	STRING	Single	Optional
`DGDBDesc`	Specifies the dynamic grammar database that contains the dynamic grammars used in the application being tuned.	STRING	Single	Optional
`Reinterpret`	Specifies whether or not the natural language value seen in deployment should be reinterpreted against the local grammars.	BOOL	Single	Optional
`SlotsToIgnore`	Specifies the list of slots that should be ignored during scoring.	STRING	Multiple	Optional
`FieldsToDump`	Specifies the call log fields that should be written out in a Records or Scores report.	STRING	Multiple	Optional
`FAInWeight`, `FRInWeight`, `CROutWeight`, `FAOutWeight`	Specify the weights applied to each of the components used to calculate the total error rate.	DOUBLE	Single	Optional
`GroupKey`	Specifies the key in the call log records used to determine the group for a record.	STRING	Single	Optional

Specifying the call log data

To specify the location of the call log data, set parameter `BaseLogDir` to the appropriate directory. You can stop at any directory level: call, year, date, etc. All the call logs under the level specified are included.

You can specify an absolute or relative path. The tools normalize the path information to start with the year. For example, consider the following paths (all valid values for parameter `BaseLogDir`):

- *c:\nuance\tuner\samples\data\PizzaTalk\logs\2003\02Feb\*
- *samples\data\PizzaTalk\logs\2003\02Feb\*

- *logs\2003\02Feb\*
- *samples\data\PizzaTalk\logs*

All the full paths to call logs and utterances under these paths (for example, *samples\data\PizzaTalk\logs\2003\02Feb\20\12\*.wav* and *samples\data\PizzaTalk\logs\2003\02Feb\20\12\*-LOG*) will be normalized to the path *2003\02Feb\20\12\.*

The directory path can contain environment variables. Set this parameter for each path you want to include.

Examples:

```
BaseLogDir %NUANCE%\tuner\samples\data\PizzaTalk\logs\
BaseLogDir c:\nuance\tuner\samples\data\PizzaTalk\logs\2003\02Feb\
BaseLogDir c:\nuance\tuner\samples\data\PizzaTalk\logs\2003\02Feb\08
```

Specifying the transcriptions files

To specify the file that contains the transcriptions, set parameter `TranscriptionsFile` to the appropriate filename. The path can contain environment variables. Set this parameter for each file you want to include.

Note: The transcriptions file must be in the standard Nuance transcription format. See Appendix E, "Nuance transcriptions conventions," for more information.

Examples:

```
TranscriptionsFile
%NUANCE%\tuner\samples\data\PizzaTalk\trans\02082003.txt
TranscriptionsFile
c:\nuance\tuner\samples\data\PizzaTalk\trans\02082003.txt
```

Specifying the recognition package

To specify the compiled recognition package used to process the data, set parameter `Package` to the appropriate directory. When running *nuance-analyze*, this package should be the package that was used in deployment. When running *nuance-tune*, you can specify any package. This parameter can contain environment variables.

Example

```
Package %NUANCE%\tuner\samples\data\PizzaTalk\grammars\MyPack
```

Specifying compilation servers

To specify how many compilation servers to start and what the command-line options for each should be, set parameter `CSArgs`. If you are using just-in-time grammars, you need to specify a compilation server to compile your grammars. There are two ways to specify it:

- Using the `CSArgs` parameter
- Through an external resource manager when starting the tools (using the `rm.Addresses` parameter on the command line). This external resource manager will point to the compilation servers required. See "Using an external resource manager" on page 160 for more information.

If you specify the compilation server using the `CSArgs` parameter, set the appropriate command-line options as a single argument included in double quotes. Make sure to include the recognition package. For example:

```
CSArgs "-package %NUANCE%\tuner\samples\data\PizzaTalk\grammars\MyPack
-config_name en-US"
```

Set the parameter for each compilation server required. These compilation servers will be started automatically. See the *Nuance API Reference* for a list of available command-line options. This parameter can contain environment variables.

Note: Parameters `comp.Port` and `rm.Addresses` should not be set in parameter `CSArgs`.

If you are using an external resource manager (specifying `rm.Addresses` on the command line) when you start the tools, parameter `CSArgs` is ignored.

If you do not use just-in-time grammars, or if you are using an external resource manager, Nuance recommends that you comment out this parameter.

Examples

```
CSArgs "-package %NUANCE%\tuner\samples\data\PizzaTalk\grammars\MyPack
-config_name en-US"
CSArgs "-package c:\data\lang\en-GB\MyPack -config_name en-GB"
```

Specifying built-in grammars

To specify the location of the built-in grammars in the system, set parameter `BuiltinContext` to the appropriate URI. This parameter corresponds to parameter `egr.builtin.context`, which is used to resolve `builtin:` grammar references.

This parameter is optional, but you must specify it if your data includes just-in-time grammars that refer to `builtin:` URIs.

Example

```
BuiltinContext http://builtinhost:8120/builtin/query/
```

Specifying the dynamic grammar database

To specify the dynamic grammar database that contains the dynamic grammars used in the application being tuned, set parameter `DGDBDesc` to the appropriate DBDescriptor. These dynamic grammars are specified in the call log data with the `DG_NUM_INSERTED` and `DG_INSERTED` fields.

Note: See the *Nuance Application Developer's Guide* for more information on the standard DBDescriptor format.

Note that this parameter does not apply for just-in-time grammars that refer to `dgdb:` URIs.

Example

```
DGDBDesc provider=fs,root=/usr/local/dgdb,name=db1,class=dgdb
```

Reinterpreting natural language values

To specify that the natural language values seen in deployment should be reinterpreted against the local grammars, set `Reinterpret` to `TRUE`. This parameter is useful for scoring if your local grammars differ from the exact grammars used during deployment. This parameter is `FALSE` by default and applies to *nuance-analyze* only.

Example

```
Reinterpret TRUE
```

Specifying slots to ignore

To specify the slots that should be ignored during scoring, set `SlotsToIgnore` to these slots. Set this parameter for each slot that should be ignored. If the NLResult from the log file differs from the natural language transcription on these slots, the utterance will still be scored as correct (if all other slots match up).

Examples

```
SlotsToIgnore prob1
SlotsToIgnore prob2
```

Specifying call log fields to include in Records and Scores reports

To specify the call log fields that should be written out in a Records or Scores report, set `FieldsToDump` to the field name. Set this parameter for each field that should be included in the report. This parameter defaults to `STATUS`.

Examples

```
FieldsToDump STATUS
FieldsToDump BEGIN_TIME
```

Changing the weights used to calculate the total error weight

To change the weights applied to each of the components used to calculate the total error rate, set the following parameters:

- `FAInWeight`: Weight of in-grammar false accept (FA) rate
- `FRInWeight`: Weight of in-grammar false reject (FR) rate
- `CROutWeight`: Weight of out-of-grammar correct reject (CR) rate
- `FAOutWeight`: Weight of out-of-grammar FA rate

(See “Scoring” on page 168 for a description of these terms.)

By default, the following values are used:

```
FAInWeight 1.0
FRInWeight 1.0
CROutWeight 0.0
FAOutWeight 1.0
```

Given these weights, the accuracy-related reports calculate the total error rate as:

$$\frac{(\texttt{FAInWeight} \times \#\ \texttt{FA-in}) + (\texttt{FRInWeight} \times \#\ \texttt{FR-in}) + (\texttt{CROutWeight} \times \#\ \texttt{CR-out}) + (\texttt{FAOutWeight} \times \#\ \texttt{FA-out})}{(\#\ \texttt{CA-in} + \#\ \texttt{FA-in} + \#\ \texttt{FR-in} + \#\ \texttt{CR-out} + \#\ \texttt{FA-out})}$$

If you want to weight each of these rates differently, set each parameter as appropriate.

Example

```
FAInWeight 1.0
FRInWeight 1.0
CROutWeight 0.0
FAOutWeight 1.0
```

Specifying how records are grouped

To specify the key in the call log records used to determine the group for a record, set `GroupKey` to this key. This parameter is set to `GRAMMAR_LABEL` by default, and backs off to `GRAMMAR` if the `GRAMMAR_LABEL` field in the call logs is not available. Change this parameter if your call log data uses a non-standard scheme to specify groupings for its grammars. See "Organizing the input data into groups" on page 144 for more information.

Example

```
GroupKey GRAMMAR_LABEL
```

Sample configuration file

The following code shows the *tuner-config* file for the sample Pizza Talk application.

```
BaseLogDir $NUANCE/tuner/samples/data/PizzaTalk/logs/

TranscriptionsFile
$NUANCE/tuner/samples/data/PizzaTalk/transcriptions/02122003.txt

Package $NUANCE/tuner/samples/data/PizzaTalk/grammars/MyPack

CSArgs "-package $NUANCE/tuner/samples/data/PizzaTalk/grammars/MyPack
-config_name en-US"
```

Running *nuance-analyze* and *nuance-tune*

The *nuance-analyze* and *nuance-tune* tools take call log data and transcriptions (optional for *nuance-analyze*) as input and generate the following output:

- Unsupervised and supervised (if transcriptions are available) reports
- Stylesheets to view the reports in HTML
- An *index.html* file, which lists the different reports generated
- A copy of the *tuner-config* configuration file used to generate the reports
- A *command-line* file, which lists the command-line options specified when running the tools
- Log files for the compilation servers, recognition servers, and resource managers started

This section shows how to use the tools most common options. It starts by describing the options that are common to both tools and then describes concepts and options specific to *nuance-analyze* and *nuance-tune*. For a detailed description of all options, see the *Nuance API Reference*.

Common options

This section describes the options that are common to both tools.

Specifying the configuration file

To specify the configuration file, use option *-config*; for example:

```
> nuance-analyze -config myConfigFile
> nuance-tune -config myConfigFile
```

If you don't specify a configuration file, the tool makes a copy of the default configuration file—*%NUANCE%\tuner\tuner-config*—and copies it to the directory where you started the tool. You can then edit this default file as required and restart the tool.

See "Editing the tuner-config file" on page 150 for more information about the configuration file.

Specifying the output directory

By default, *nuance-analyze* stores the reports and related output files in a subdirectory called *analysis*, while *nuance-tune* stores the reports and related output files in a subdirectory called *experiment* (in the directory where the tool was started). To specify a different directory, use option *-o*; for example:

```
> nuance-analyze -config myConfigFile -o analysisTest1
> nuance-tune -config myConfigFile -o experimentTest1
```

If the output data already exists when you run the tools, it is not overwritten automatically. The tools return an error message saying that the directory already exists. To overwrite the data in the directory, use option *-force*; for example:

```
> nuance-analyze -config myConfigFile -o analysisTest1 -force
```

Filtering data

By default, all the call logs are processed when you run the tools. To process a subset of the call log data available, you can use the *-begin_date*, *-end_date*, *-group*, and *-dataset* options; for example:

```
> nuance-analyze -config myConfigFile -begin_date "Jan 28 2003"
-end_date "Jan 30 2003"
```

This command will process all the calls from January 28 (inclusive) to January 30 (inclusive). Here is another example:

```
> nuance-tune -config myConfigFile -begin_date "Jan 28 2003"
-group GetPizzaToppings
```

This command will process the utterances in the group GetPizzaToppings for all the calls after January 28 (inclusive).

You can also specify a dataset file that lists the calls and utterances to process; for example:

```
> nuance-tune -config myConfigFile -dataset dev.xml
```

See “Filtering call log data” on page 145 for more information.

Specifying a grammar overrides file

To specify the grammars and parameters that should be overridden for this analysis, use option *-grammar_overrides*; for example:

```
> nuance-analyze -config myConfigFile -grammar_overrides
Test1overrides.xml
```

See “Specifying grammar overrides” on page 147 for more information on creating the grammar overrides file.

Running *nuance-analyze*

The *nuance-analyze* tool lets you generate the Records report, which provides the recognition results for all the call log records in a call. This report is useful for analyzing data before transcriptions are available, to help identify and investigate potential problem areas flagged by the Status report, and so on. For more information about the Records report, see “Working with the Records and Scores reports” on page 176.

To generate the Records report, use option *-dump_records*; for example:

```
> nuance-analyze -config myConfigFile -grammar_overrides
Test1overrides.xml -dump_records
```

By default, only the STATUS field from the call log is saved in the Records report. To include other fields, set the *tuner-config* parameter `FieldsToDump` for each field to include. See “Specifying call log fields to include in Records and Scores reports” on page 156 for more information.

Running *nuance-tune*

This section describes concepts specific to *nuance-tune*.

Setting recognition parameters for the recognition experiments

When you run recognition experiments, *nuance-tune* sets parameters according to the order of precedence described in “Priority of parameter settings” on page 148. You can view the parameters used to run the recognition experiments by looking at the Groups report.

In all cases, the following “frozen” parameters are set to the values specified below and are not overwritten, regardless of the values seen in the deployed application or specified in a grammar overrides file. These values are required for the Rejection and N-best reports to be meaningful.

Parameter	Value
rec.ConfidenceRejectionThreshold	-1
rec.DoNBest	TRUE

Although the value of parameter `rec.ConfidenceRejectionThreshold` is frozen to -1 when you run *nuance-tune*, all the *nuance-tune* reports (except for Rejection Threshold) are based on the intended value of `rec.ConfidenceRejectionThreshold`. For example, if `rec.ConfidenceRejectionThreshold` was specified in the grammar overrides file or in a static context, that value will not be used in *nuance-tune* experiments, but *nuance-tune* will manually reject the utterances based on that value (dropping back to a default of 45). *nuance-tune* will also use the scoring based on manual rejection for all of its reports, except for the Rejection Threshold report. The Rejection Threshold report sweeps through all the `rec.ConfidenceRejectionThreshold` values and produces accuracy statistics for all values from 0 to 100.

Experiment options: Guidelines

Follow these guidelines when trying out different experiment options:

For static grammar-based applications: To tweak both the grammars and the recognition parameters used in your recognition experiments, compile your own version of the static package and point to your version using the `Package` parameter in your *tuner-config* file. However, if you simply want to try a new set of parameters, leave the package untouched and specify the new parameters (without touching the grammars) in a grammar overrides file, using the *-grammar_overrides* command-line option.

For just-in-time grammar-based applications: To try different grammars and parameters, leave the package untouched (unless you're modifying the contents of a `<static:...>` reference) and specify the experiment grammar using the grammar overrides file. You might also want to do this, for example, if you want to rewrite the just-in-time grammars to point to your own set of grammar sources.

Using an external resource manager

By default, *nuance-tune* starts the processes—resource managers, recognition servers, etc.—required to perform recognition experiments. However, if you want some control over these processes for your experiments—for example, you want to distribute a set of *nuance-tune* jobs but want to use a central set of recognition servers for all of them—you can specify an external resource manager that points to its own set of registered servers.

To specify an external resource manager, use parameter `rm.Addresses` on the command line. For example:

```
> nuance-tune -config myConfigFile -o analysisTest1 -force
rm.Addresses=machine1
```

Looking at reports

As described earlier, there are two types of reports. *Unsupervised reports* are based on raw call log data, without transcriptions, while *supervised reports* are based on call log data (possibly with new recognition results from an offline experiment) and on transcriptions. The following table lists the supervised and unsupervised reports generated by the tools.

Report type	Report name	nuance-analyze	nuance-tune
Unsupervised	Call Volume	X	
	Latency	X	
	Speech Duration	X	
	XCPURT		X
	Groups	X	X
	Status (Hot Spot)	X	X
Supervised	Accuracy	X	X
	N-best	X	X
	Rejection Threshold	X	X
	IG (in-grammar)	X	X
	OOG (out-of-grammar)	X	X
	OOC (out-of-coverage)	X	X
	Error	X	X
	Confusion	X	X

The reports are output as XML documents, which can then be transformed into formats such as HTML, PDF, RTF, and so on. By default, Nuance provides stylesheets that let you see the reports in HTML in Internet Explorer 6.0. The tools also generate an *index.html* file, which references all the reports generated.

This section describes each supervised and unsupervised report. It also describes how to convert the XML reports to another format for viewing.

Unsupervised reports

Unsupervised reports are based on raw call log data, without transcriptions. These reports can provide insight into which parts of an application seem to be working well, as well as which parts might warrant deeper investigation (with transcriptions).

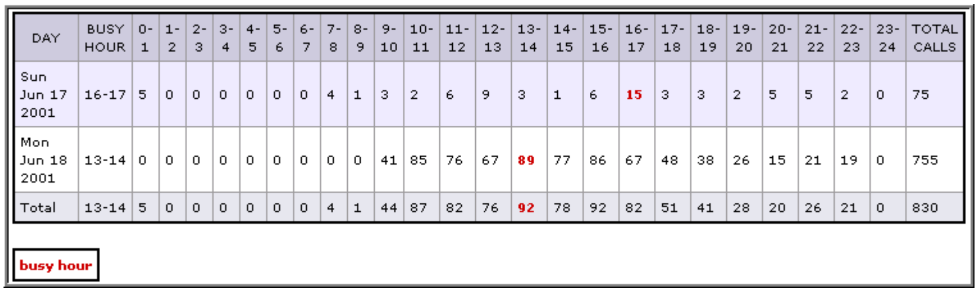
The Analysis and Tuning Tools generate the following unsupervised reports:

Call Volume

This report provides the number of calls handled by the application, organized per hour and per day. As shown below, the Call Volume report contains the following information:

- Number of calls per hour for all days included in the call log data, with totals per day and per hour
- Busiest hour for each day; this peak hour is also highlighted in red in the report

Note: This report is generated with *nuance-analyze* only.



DAY	BUSY HOUR	0-1	1-2	2-3	3-4	4-5	5-6	6-7	7-8	8-9	9-10	10-11	11-12	12-13	13-14	14-15	15-16	16-17	17-18	18-19	19-20	20-21	21-22	22-23	23-24	TOTAL CALLS
Sun Jun 17 2001	16-17	5	0	0	0	0	0	0	4	1	3	2	6	9	3	1	6	**15**	3	3	2	5	5	2	0	75
Mon Jun 18 2001	13-14	0	0	0	0	0	0	0	0	0	41	85	76	67	**89**	77	86	67	48	38	26	15	21	19	0	755
Total	13-14	5	0	0	0	0	0	0	4	1	44	87	82	76	**92**	78	92	82	51	41	28	20	26	21	0	830

busy hour

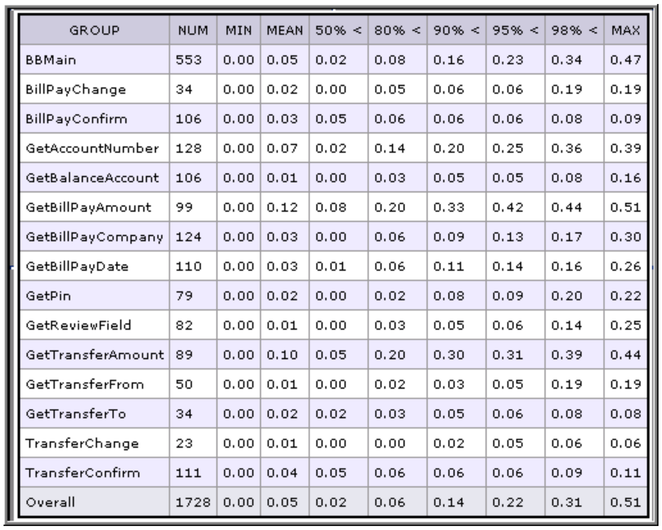
Latency

This report lists for each group the latency (in seconds) from the time end-of-speech is detected in the input utterance until the final recognition result arrives. As shown below, the Latency report contains the following information:

- **NUM**: Number of utterances in the group
- **MIN**: Minimum latency observed within the group
- **MEAN**: Average latency for the group
- **MAX**: Maximum latency observed within the group
- **50%, 80%, 90%, 95%, 98%**: Percentile statistics on the latency values; for example, the 80th percentile column indicates that 80% of the utterances in the group have the latency value specified or less

If a latency value greater than 2 seconds is observed in the 98th percentile column for a group, the group name is highlight in red to indicate a potential problem area.

Note: This report is generated with *nuance-analyze* only.



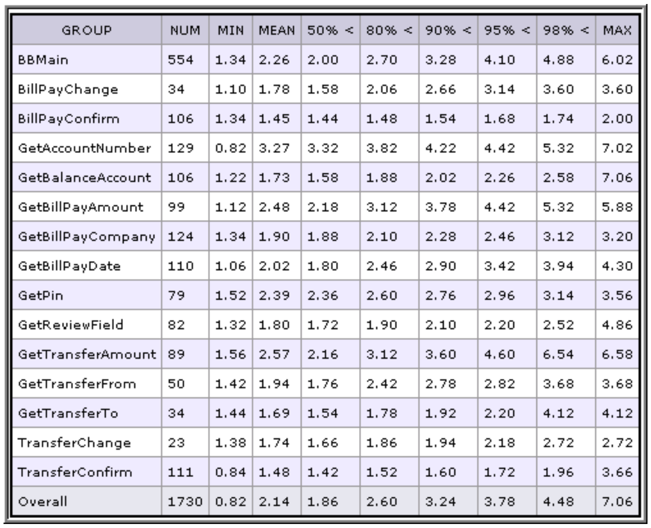
GROUP	NUM	MIN	MEAN	50% <	80% <	90% <	95% <	98% <	MAX
BBMain	553	0.00	0.05	0.02	0.08	0.16	0.23	0.34	0.47
BillPayChange	34	0.00	0.02	0.00	0.05	0.06	0.06	0.19	0.19
BillPayConfirm	106	0.00	0.03	0.05	0.06	0.06	0.06	0.08	0.09
GetAccountNumber	128	0.00	0.07	0.02	0.14	0.20	0.25	0.36	0.39
GetBalanceAccount	106	0.00	0.01	0.00	0.03	0.05	0.05	0.08	0.16
GetBillPayAmount	99	0.00	0.12	0.08	0.20	0.33	0.42	0.44	0.51
GetBillPayCompany	124	0.00	0.03	0.00	0.06	0.09	0.13	0.17	0.30
GetBillPayDate	110	0.00	0.03	0.01	0.06	0.11	0.14	0.16	0.26
GetPin	79	0.00	0.02	0.00	0.02	0.08	0.09	0.20	0.22
GetReviewField	82	0.00	0.01	0.00	0.03	0.05	0.06	0.14	0.25
GetTransferAmount	89	0.00	0.10	0.05	0.20	0.30	0.31	0.39	0.44
GetTransferFrom	50	0.00	0.01	0.00	0.02	0.03	0.05	0.19	0.19
GetTransferTo	34	0.00	0.02	0.02	0.03	0.05	0.06	0.08	0.08
TransferChange	23	0.00	0.01	0.00	0.00	0.02	0.05	0.06	0.06
TransferConfirm	111	0.00	0.04	0.05	0.06	0.06	0.06	0.09	0.11
Overall	1728	0.00	0.05	0.02	0.06	0.14	0.22	0.31	0.51

Speech Duration

This report lists the `SPEECH_DURATION` values seen for the records within a group. This value is the length of the utterance (in seconds) sent to the recognizer. As shown below, the Speech Duration report contains the following information:

- **NUM**: Number of utterances in the group
- **MIN**: Minimum utterance length observed within the group
- **MEAN**: Average utterance length for the group
- **MAX**: Maximum utterance length observed within the group
- **50%**, **80%**, **90%**, **95%**, **98%**: Percentile statistics on the speech duration values; for example, the 80th percentile column indicates that 80% of the utterances in the group have the utterance length specified or less

Note: This report is generated with *nuance-analyze* only.



GROUP	NUM	MIN	MEAN	50% <	80% <	90% <	95% <	98% <	MAX
BBMain	554	1.34	2.26	2.00	2.70	3.28	4.10	4.88	6.02
BillPayChange	34	1.10	1.78	1.58	2.06	2.66	3.14	3.60	3.60
BillPayConfirm	106	1.34	1.45	1.44	1.48	1.54	1.68	1.74	2.00
GetAccountNumber	129	0.82	3.27	3.32	3.82	4.22	4.42	5.32	7.02
GetBalanceAccount	106	1.22	1.73	1.58	1.88	2.02	2.26	2.58	7.06
GetBillPayAmount	99	1.12	2.48	2.18	3.12	3.78	4.42	5.32	5.88
GetBillPayCompany	124	1.34	1.90	1.88	2.10	2.28	2.46	3.12	3.20
GetBillPayDate	110	1.06	2.02	1.80	2.46	2.90	3.42	3.94	4.30
GetPin	79	1.52	2.39	2.36	2.60	2.76	2.96	3.14	3.56
GetReviewField	82	1.32	1.80	1.72	1.90	2.10	2.20	2.52	4.86
GetTransferAmount	89	1.56	2.57	2.16	3.12	3.60	4.60	6.54	6.58
GetTransferFrom	50	1.42	1.94	1.76	2.42	2.78	2.82	3.68	3.68
GetTransferTo	34	1.44	1.69	1.54	1.78	1.92	2.20	4.12	4.12
TransferChange	23	1.38	1.74	1.66	1.86	1.94	2.18	2.72	2.72
TransferConfirm	111	0.84	1.48	1.42	1.52	1.60	1.72	1.96	3.66
Overall	1730	0.82	2.14	1.86	2.60	3.24	3.78	4.48	7.06

XCPURT

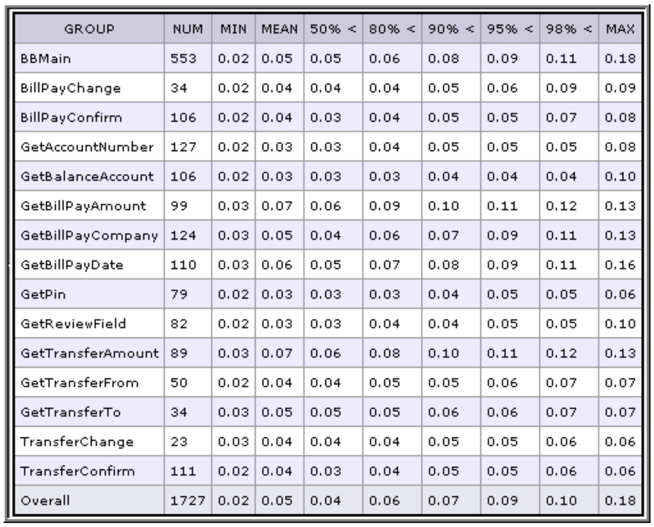
The XCPURT value is a measure of the CPU required to recognize the utterance. A value of 0.5 implies that the recognition task requires 50% of the CPU to process the utterance in real time (or, equivalently, that two simultaneous recognitions can take place without the system falling behind).

This report lists the XCPURT values seen for the recognition results (RecResult) within a group when running recognition experiments. Note that this report is only meaningful if only one channel of recognition is active at a time in the recserver; the experiment must be a controlled one.

As shown below, the XCPURT report contains the following information:

- **NUM**: Number of utterances in the group
- **MIN**: Minimum XCPURT value observed within the group
- **MEAN**: Average XCPURT value for the group
- **MAX**: Maximum XCPURT value observed within the group
- **50%, 80%, 90%, 95%, 98%**: Percentile statistics on the XCPURT values; for example, the 80th percentile column indicates that 80% of the recognition results in the group have the XCPURT value specified or less

Note: This report is generated with *nuance-tune* only.



GROUP	NUM	MIN	MEAN	50% <	80% <	90% <	95% <	98% <	MAX
BBMain	553	0.02	0.05	0.05	0.06	0.08	0.09	0.11	0.18
BillPayChange	34	0.02	0.04	0.04	0.04	0.05	0.06	0.09	0.09
BillPayConfirm	106	0.02	0.04	0.03	0.04	0.05	0.05	0.07	0.08
GetAccountNumber	127	0.02	0.03	0.03	0.04	0.05	0.05	0.05	0.08
GetBalanceAccount	106	0.02	0.03	0.03	0.03	0.04	0.04	0.04	0.10
GetBillPayAmount	99	0.03	0.07	0.06	0.09	0.10	0.11	0.12	0.13
GetBillPayCompany	124	0.03	0.05	0.04	0.06	0.07	0.09	0.11	0.13
GetBillPayDate	110	0.03	0.06	0.05	0.07	0.08	0.09	0.11	0.16
GetPin	79	0.02	0.03	0.03	0.03	0.04	0.05	0.05	0.06
GetReviewField	82	0.02	0.03	0.03	0.04	0.04	0.05	0.05	0.10
GetTransferAmount	89	0.03	0.07	0.06	0.08	0.10	0.11	0.12	0.13
GetTransferFrom	50	0.02	0.04	0.04	0.05	0.05	0.06	0.07	0.07
GetTransferTo	34	0.03	0.05	0.05	0.05	0.06	0.06	0.07	0.07
TransferChange	23	0.03	0.04	0.04	0.04	0.05	0.05	0.06	0.06
TransferConfirm	111	0.02	0.04	0.03	0.04	0.05	0.05	0.06	0.06
Overall	1727	0.02	0.05	0.04	0.06	0.07	0.09	0.10	0.18

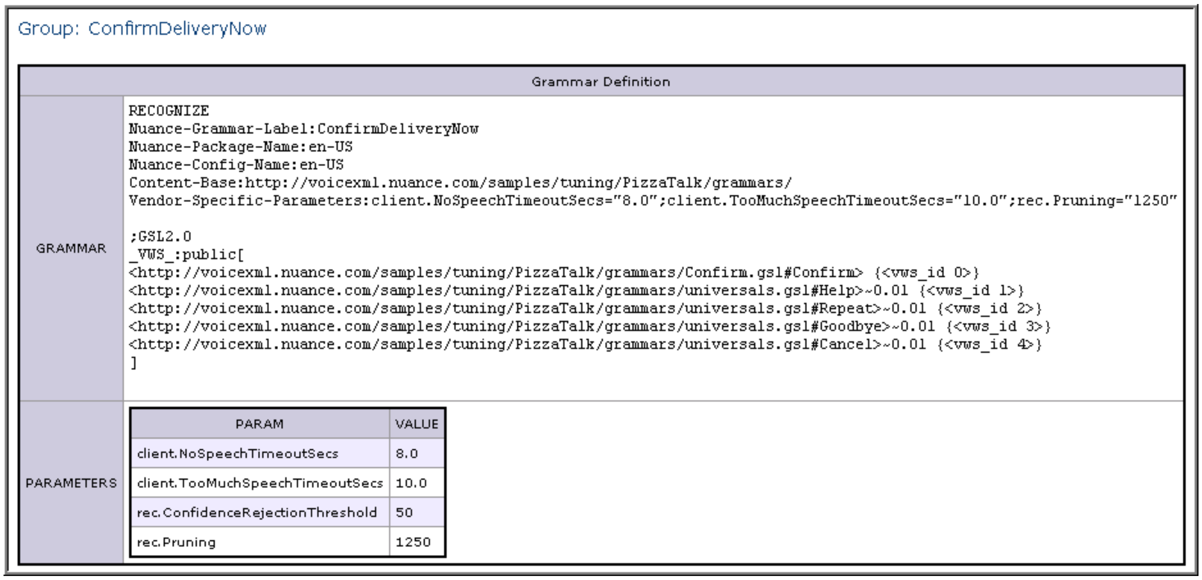
Groups

This report lists the grammars and parameters used for each group. If applicable, the report provides a table listing the default parameter values set for the application. Then for each group it provides:

- For static grammars, the name of the grammar
- For just-in-time grammars, the just-in-time request
- For dynamic grammars, the name of the top-level static grammar as well as all dynamic grammar insertions
- If applicable, parameter values set for the recognition request

Note: When you run this report, an instance identifier is assigned to each unique grammar within a group. This identifier is useful when you want to override an individual instance of a grammar within a group. See “Grammar instances” on page 150 for more information. If there is just one grammar definition in a group, the instance identifier “default” is used.

The following graphic shows an except from a Groups report:


Group: ConfirmDeliveryNow

Grammar Definition

GRAMMAR

```
RECOGNIZE
Nuance-Grammar-Label:ConfirmDeliveryNow
Nuance-Package-Name:en-US
Nuance-Config-Name:en-US
Content-Base:http://voicexml.nuance.com/samples/tuning/PizzaTalk/grammars/
Vendor-Specific-Parameters:client.NoSpeechTimeoutSecs="8.0";client.TooMuchSpeechTimeoutSecs="10.0";rec.Pruning="1250"

;GSL2.0
_VWS_:public[
<http://voicexml.nuance.com/samples/tuning/PizzaTalk/grammars/Confirm.gsl#Confirm> {<vws_id 0>}
<http://voicexml.nuance.com/samples/tuning/PizzaTalk/grammars/universals.gsl#Help>~0.01 {<vws_id 1>}
<http://voicexml.nuance.com/samples/tuning/PizzaTalk/grammars/universals.gsl#Repeat>~0.01 {<vws_id 2>}
<http://voicexml.nuance.com/samples/tuning/PizzaTalk/grammars/universals.gsl#Goodbye>~0.01 {<vws_id 3>}
<http://voicexml.nuance.com/samples/tuning/PizzaTalk/grammars/universals.gsl#Cancel>~0.01 {<vws_id 4>}
]
```

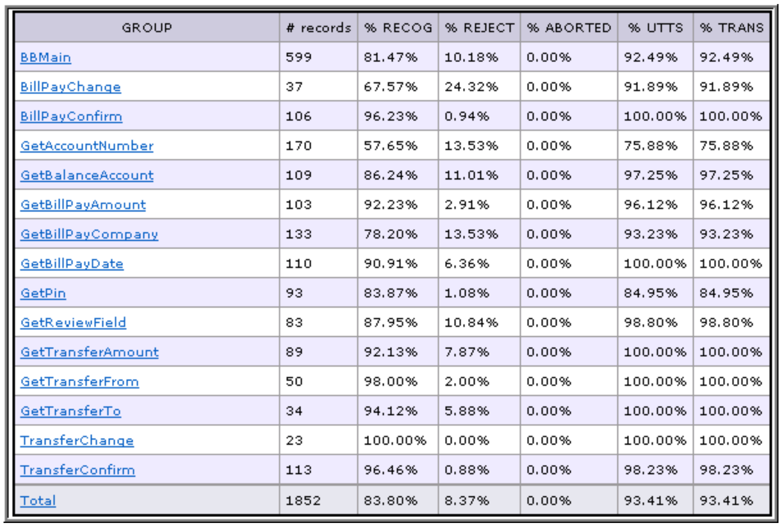
PARAMETERS

PARAM	VALUE
client.NoSpeechTimeoutSecs	8.0
client.TooMuchSpeechTimeoutSecs	10.0
rec.ConfidenceRejectionThreshold	50
rec.Pruning	1250

Status (Hot Spot)

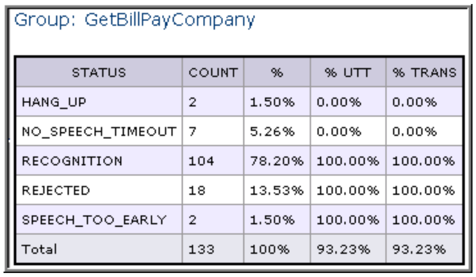
This report provides the percentage of calls that resulted in a status of RECOGNITION, REJECTED, and ABORTED, organized per group. This report is very useful for quickly spotting problems in an application. As shown below, the report includes the following information:

- Number of records in the group
- Percentage of records in the group with a status of RECOGNITION, REJECTED, and ABORTED.
- Percentage of records that have recorded utterances
- Percentage of records that have transcriptions for their logged utterances (if transcriptions are available)



GROUP	# records	% RECOG	% REJECT	% ABORTED	% UTTS	% TRANS
BBMain	599	81.47%	10.18%	0.00%	92.49%	92.49%
BillPayChange	37	67.57%	24.32%	0.00%	91.89%	91.89%
BillPayConfirm	106	96.23%	0.94%	0.00%	100.00%	100.00%
GetAccountNumber	170	57.65%	13.53%	0.00%	75.88%	75.88%
GetBalanceAccount	109	86.24%	11.01%	0.00%	97.25%	97.25%
GetBillPayAmount	103	92.23%	2.91%	0.00%	96.12%	96.12%
GetBillPayCompany	133	78.20%	13.53%	0.00%	93.23%	93.23%
GetBillPayDate	110	90.91%	6.36%	0.00%	100.00%	100.00%
GetPin	93	83.87%	1.08%	0.00%	84.95%	84.95%
GetReviewField	83	87.95%	10.84%	0.00%	98.80%	98.80%
GetTransferAmount	89	92.13%	7.87%	0.00%	100.00%	100.00%
GetTransferFrom	50	98.00%	2.00%	0.00%	100.00%	100.00%
GetTransferTo	34	94.12%	5.88%	0.00%	100.00%	100.00%
TransferChange	23	100.00%	0.00%	0.00%	100.00%	100.00%
TransferConfirm	113	96.46%	0.88%	0.00%	98.23%	98.23%
Total	1852	83.80%	8.37%	0.00%	93.41%	93.41%

For each group, the report then provides the number and percentage of records with each type of status seen in deployment. Again, for each status, the report provides the percentage of records that have recorded utterances and, if transcriptions are available, the percentage of records that have transcriptions for their logged utterances.



Group: GetBillPayCompany

STATUS	COUNT	%	% UTT	% TRANS
HANG_UP	2	1.50%	0.00%	0.00%
NO_SPEECH_TIMEOUT	7	5.26%	0.00%	0.00%
RECOGNITION	104	78.20%	100.00%	100.00%
REJECTED	18	13.53%	100.00%	100.00%
SPEECH_TOO_EARLY	2	1.50%	100.00%	100.00%
Total	133	100%	93.23%	93.23%

Some notes:

- Records that are thrown out—see "Supervised reports" on page 168 for more information—are not counted in the % TRANS column, even if a transcription is available.
- If produced by *nuance-tune*, this report only includes records that were processed and thus had transcriptions; therefore, the %UTT and %TRANS values will always be 100%. In this case, this report primarily provides information on which utterances are recognized vs. rejected.

Supervised reports

Supervised reports are based both on call log data and on transcriptions. Note that the transcriptions must follow the standard Nuance transcription conventions. See Appendix E, “Nuance transcriptions conventions,” for more information.

Supervised reports are generated as follows. The Analysis and Tuning Tools load all available transcriptions and attempt to match these transcriptions with the utterances in the call log data. The tools then extract the subset of call log records that meet the following criteria:

- They have an utterance and a transcription for that utterance
- They have a status of RECOGNITION, REJECTED, or RECOGNIZER_TOO_SLOW_TIMEOUT
- The transcription for the utterance does not include “dtmf”, “hang_up”, or “hangup”.

The tools then generate natural language references from the transcriptions for the subset of data meeting these criteria.

Note: The grammars used in deployment must be available when you run the tools. You can also use the *-grammar_overrides* command-line option to point the tools to local copies of these grammars, if necessary.

Scoring

Once the natural language references have been generated, the tools score each record against the natural language reference. The scoring process involves assessing:

- whether the natural language reference is in grammar or out of grammar
- whether the recognition result matches this assessment

  Note: If the natural language reference has multiple interpretations (for example, ambiguous interpretations), all the interpretations in the natural language reference must match all the interpretations in the recognition result.

nuance-analyze scores the recognition result seen in deployment, while *nuance-tune* scores a fresh recognition result produced by rerecognizing the logged utterance.

Note: When running *nuance-analyze*, if your local grammars do not match the ones used in the deployed application, you may want to set the `Reinterpret` *tuner-config* parameter to `TRUE`. The recognition results obtained in deployment are then reinterpreted against your local grammars. If the recognition results obtained in deployment are no longer in grammar, the RecResult is flagged as REJECTED.

During the scoring process, the records are marked with the following accuracy labels:

- **IG** (in-grammar utterance): The caller said a word or phrase that was parsed by the grammar.
- **OOG** (out-of-grammar utterance): The caller said a word or phrase that could not be parsed by the grammar.
- **OOC** (out-of-coverage utterance): The subset of out-of-grammar data that does not contain any fragments, incomprehensible speech, or other disfluencies that no grammar can cover. The OOC data represents the OOG utterances that can be reasonably addressed by updating the grammar.
- **CA-in (**in-grammar correct accept): The utterance was in-grammar and the recognizer correctly understood what the caller said.
- **FA-in**: (in-grammar false accept): The utterance was in-grammar and the recognizer accepted the utterance, but it recognized it incorrectly (for example, a misrecognition/confusion).
- **FR-in** (in-grammar false reject): The utterance was in-grammar, but the recognizer rejected the utterance.
- **CR-out** (out-of-grammar correct reject): The utterance was out-of-grammar and the recognizer correctly rejected the utterance.
- **FA-out** (out-of-grammar false accept): The utterance was out-of-grammar and the recognizer accepted the utterance, but it should have been rejected.

If the natural language reference is in grammar and the recognition result had a status of RECOGNITION, the actual result is examined to see if any of its hypotheses match the natural language reference (ignoring any slots specified in the `SlotsToIgnore` *tuner-config* parameter). If the top hypothesis matches, the record is marked CA-in; otherwise, the record is marked FA-in. If the recognition result was anything other than RECOGNITION, the record is marked FR-in.

If the natural language reference is out of grammar and the recognition result had a status of RECOGNITION, the record is marked FA-out. If the recognition result was anything other than RECOGNITION, the record is marked CR-out.

The following table summarizes how the utterances are marked:

NL reference	Recognition result	Accuracy label
in-grammar	RECOGNITION	**CA-in**: If top hypothesis matches NL ref **FA-in**: If top hypothesis doesn't match
	not RECOGNITION	**FR-in**
out-of-grammar	RECOGNITION	**FA-out**
	not RECOGNITION	**CR-out**

Out-of-coverage data

Because some out-of-grammar data contain fragments, incomprehensible speech, and other disfluencies that no grammar will be able to cover, the tools further classify a subset of the out-of-grammar data as out-of-coverage; grammar tuning can help out-of-coverage performance. An out-of-grammar utterance is considered out of coverage if its transcription does not contain the `[fragment]` or `(())` marker, and some speech content is left after filtering out markers such as `[side_speech]`.

Total error rate

Given the number of CA-in, FA-in, FR-in, CR-out, FA-out utterances for each group (represented by `# CA-in`, `# FA-in`, etc., in the formula below), the tools compute a total error statistic as:

$$\frac{(\texttt{FAInWeight x \# FA-in}) + (\texttt{FRInWeight x \# FR-in}) + (\texttt{CROutWeight x \# CR-out}) + (\texttt{FAOutWeight x \# FA-out})}{(\texttt{\# CA-in + \# FA-in + \# FR-in + \# CR-out + \# FA-out})}$$

By default, weights of 1.0 are used for `FAInWeight`, `FRInWeight`, and `FAOutWeight`, and a weight of 0.0 is used for `CROutWeight`. These weights can be configured in the *tuner-config* file. Most of the supervised reports are based on these scores. See "Changing the weights used to calculate the total error weight" on page 156 for more information.

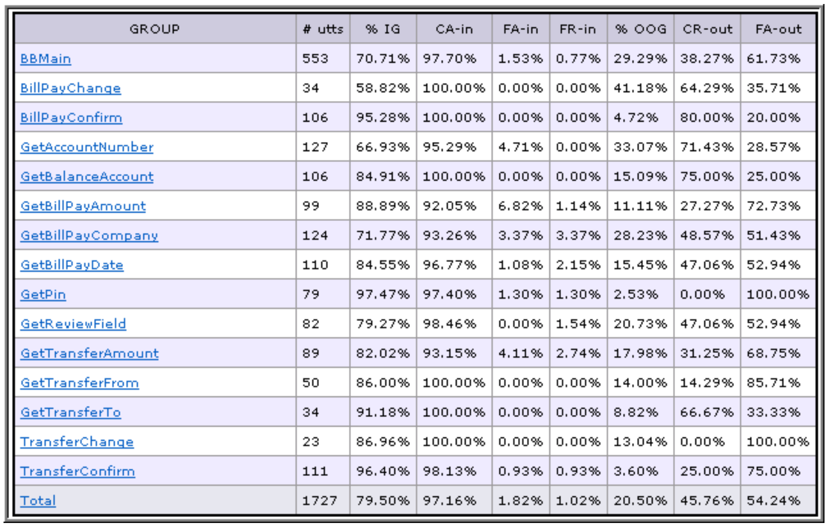
The Analysis and Tuning Tools generate the following supervised reports:

Accuracy

As shown below, this report provides the following summary accuracy statistics per group:

- **# utt:** number of utterances in the group
- **%IG**: percentage of in-grammar utterances
- **CA-in**: in-grammar correct accept rate
- **FA-in**: in-grammar false accept rate
- **FR-in**: in-grammar false reject rate

- **%OOG**: percentage of out-of-grammar utterances (including out-of-coverage)
- **CR-out**: out-of-grammar correct reject rate
- **FA-out**: out-of-grammar false accept rate

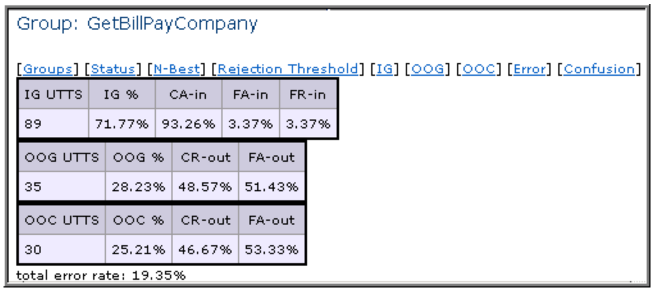


GROUP	# utts	% IG	CA-in	FA-in	FR-in	% OOG	CR-out	FA-out
BBMain	553	70.71%	97.70%	1.53%	0.77%	29.29%	38.27%	61.73%
BillPayChange	34	58.82%	100.00%	0.00%	0.00%	41.18%	64.29%	35.71%
BillPayConfirm	106	95.28%	100.00%	0.00%	0.00%	4.72%	80.00%	20.00%
GetAccountNumber	127	66.93%	95.29%	4.71%	0.00%	33.07%	71.43%	28.57%
GetBalanceAccount	106	84.91%	100.00%	0.00%	0.00%	15.09%	75.00%	25.00%
GetBillPayAmount	99	88.89%	92.05%	6.82%	1.14%	11.11%	27.27%	72.73%
GetBillPayCompany	124	71.77%	93.26%	3.37%	3.37%	28.23%	48.57%	51.43%
GetBillPayDate	110	84.55%	96.77%	1.08%	2.15%	15.45%	47.06%	52.94%
GetPin	79	97.47%	97.40%	1.30%	1.30%	2.53%	0.00%	100.00%
GetReviewField	82	79.27%	98.46%	0.00%	1.54%	20.73%	47.06%	52.94%
GetTransferAmount	89	82.02%	93.15%	4.11%	2.74%	17.98%	31.25%	68.75%
GetTransferFrom	50	86.00%	100.00%	0.00%	0.00%	14.00%	14.29%	85.71%
GetTransferTo	34	91.18%	100.00%	0.00%	0.00%	8.82%	66.67%	33.33%
TransferChange	23	86.96%	100.00%	0.00%	0.00%	13.04%	0.00%	100.00%
TransferConfirm	111	96.40%	98.13%	0.93%	0.93%	3.60%	25.00%	75.00%
Total	1727	79.50%	97.16%	1.82%	1.02%	20.50%	45.76%	54.24%

For each group, the report also provides the following information:

- Number and percentage of utterances that are in grammar, out of grammar, and out of coverage, as well as other accuracy statistics
- Total error rate for the top hypothesis for each record in the group
- Links to other reports for the same group.

For example:



Group: GetBillPayCompany

[Groups] [Status] [N-Best] [Rejection Threshold] [IG] [OOG] [OOC] [Error] [Confusion]

IG UTTS	IG %	CA-in	FA-in	FR-in
89	71.77%	93.26%	3.37%	3.37%

OOG UTTS	OOG %	CR-out	FA-out
35	28.23%	48.57%	51.43%

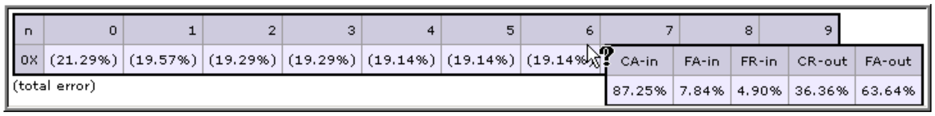
OOC UTTS	OOC %	CR-out	FA-out
30	25.21%	46.67%	53.33%

total error rate: 19.35%

N-best

This report provides the total error rate, displayed in parenthesis, for different N-best values, from 1 to the maximum number of hypotheses seen for that group. It also includes more detailed accuracy statistics for each N-best value

(CA-in, FA-in, FR-in, CR-out, FA-out). The default stylesheet displays these values as a “tool tip” when you move your cursor over an N-best value.



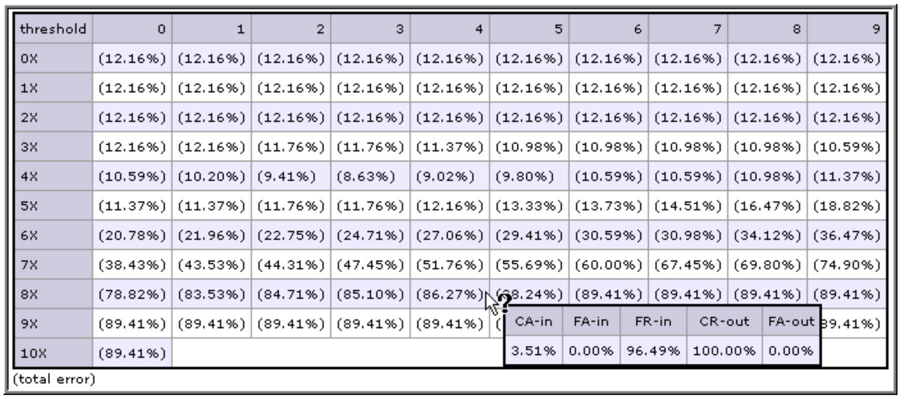
n	0	1	2	3	4	5	6	7	8	9
0X	(21.29%)	(19.57%)	(19.29%)	(19.29%)	(19.14%)	(19.14%)	(19.14%			

(total error)

CA-in	FA-in	FR-in	CR-out	FA-out
87.25%	7.84%	4.90%	36.36%	63.64%

Rejection Threshold This report provides the total error rate for different rejection thresholds, from 0 to 100, inclusive. It also includes more detailed accuracy statistics for each threshold (CA-in, FA-in, FR-in, CR-out, FA-out). The default stylesheet displays these values as a “tool tip” when you move your cursor over a threshold.

Note: This report is only complete when `rec.ConfidenceRejectionThreshold=-1`, which is set automatically when you run *nuance-tune*.

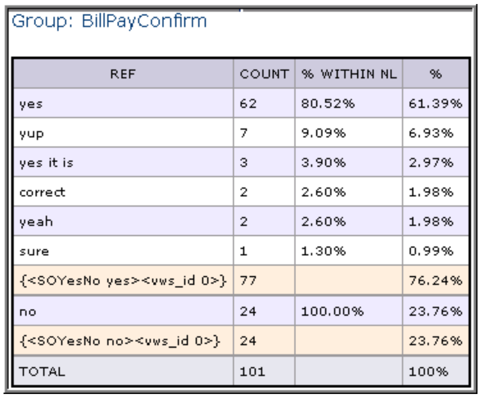


threshold	0	1	2	3	4	5	6	7	8	9
0X	(12.16%)	(12.16%)	(12.16%)	(12.16%)	(12.16%)	(12.16%)	(12.16%)	(12.16%)	(12.16%)	(12.16%)
1X	(12.16%)	(12.16%)	(12.16%)	(12.16%)	(12.16%)	(12.16%)	(12.16%)	(12.16%)	(12.16%)	(12.16%)
2X	(12.16%)	(12.16%)	(12.16%)	(12.16%)	(12.16%)	(12.16%)	(12.16%)	(12.16%)	(12.16%)	(12.16%)
3X	(12.16%)	(12.16%)	(11.76%)	(11.76%)	(11.37%)	(10.98%)	(10.98%)	(10.98%)	(10.98%)	(10.59%)
4X	(10.59%)	(10.20%)	(9.41%)	(8.63%)	(9.02%)	(9.80%)	(10.59%)	(10.59%)	(10.98%)	(11.37%)
5X	(11.37%)	(11.37%)	(11.76%)	(11.76%)	(12.16%)	(13.33%)	(13.73%)	(14.51%)	(16.47%)	(18.82%)
6X	(20.78%)	(21.96%)	(22.75%)	(24.71%)	(27.06%)	(29.41%)	(30.59%)	(30.98%)	(34.12%)	(36.47%)
7X	(38.43%)	(43.53%)	(44.31%)	(47.45%)	(51.76%)	(55.69%)	(60.00%)	(67.45%)	(69.80%)	(74.90%)
8X	(78.82%)	(83.53%)	(84.71%)	(85.10%)	(86.27%)	8.24%)	(89.41%)	(89.41%)	(89.41%)	(89.41%)
9X	(89.41%)	(89.41%)	(89.41%)	(89.41%)	(89.41%)	(				39.41%)
10X	(89.41%)									

(total error)

CA-in	FA-in	FR-in	CR-out	FA-out
3.51%	0.00%	96.49%	100.00%	0.00%

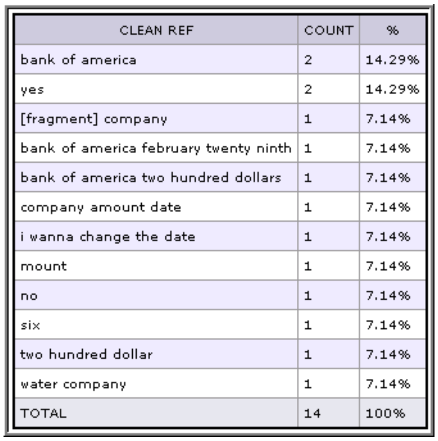
IG (in-grammar) This report provides the most common in-grammar transcriptions, sorted by frequency of occurrence, for each natural language reference in the group. For each type of in-grammar transcription, it also provides the percentage of records with that utterance within the natural language reference and within the group.


Group: BillPayConfirm

REF	COUNT	% WITHIN NL	%
yes	62	80.52%	61.39%
yup	7	9.09%	6.93%
yes it is	3	3.90%	2.97%
correct	2	2.60%	1.98%
yeah	2	2.60%	1.98%
sure	1	1.30%	0.99%
{<SOYesNo yes><vws_id 0>}	77		76.24%
no	24	100.00%	23.76%
{<SOYesNo no><vws_id 0>}	24		23.76%
TOTAL	101		100%

OOG (out-of-grammar)

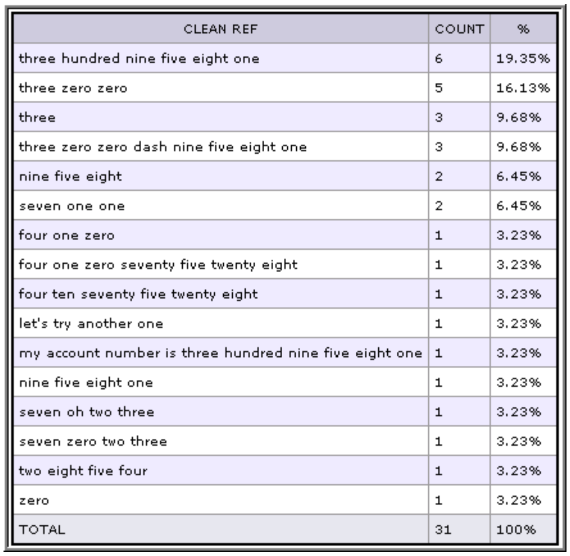
This report provides the most common out-of-grammar transcriptions, sorted by frequency of occurrence. The transcriptions are clean in that all the markers are removed except for `[fragment]` and `(())`. Note that utterances with an empty transcription (after cleaning) appear in the OOG report as `[empty]`.



CLEAN REF	COUNT	%
bank of america	2	14.29%
yes	2	14.29%
[fragment] company	1	7.14%
bank of america february twenty ninth	1	7.14%
bank of america two hundred dollars	1	7.14%
company amount date	1	7.14%
i wanna change the date	1	7.14%
mount	1	7.14%
no	1	7.14%
six	1	7.14%
two hundred dollar	1	7.14%
water company	1	7.14%
TOTAL	14	100%

OOC (out-of-coverage)

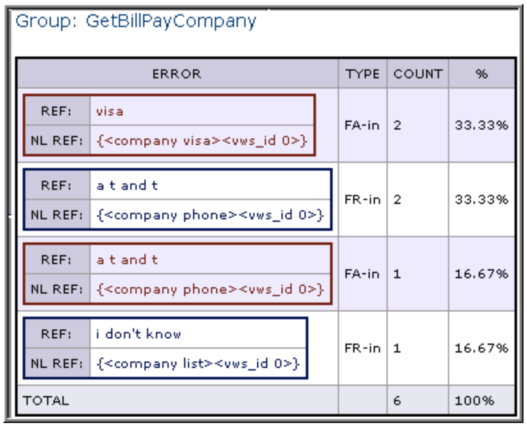
This report provides the most common out-of-coverage transcriptions, sorted by frequency of occurrence. This is the subset of out-of-coverage transcriptions that can be improved through grammar tuning. The transcriptions are clean in that all the markers are removed except for `[fragment]` and `(())`.



CLEAN REF	COUNT	%
three hundred nine five eight one	6	19.35%
three zero zero	5	16.13%
three	3	9.68%
three zero zero dash nine five eight one	3	9.68%
nine five eight	2	6.45%
seven one one	2	6.45%
four one zero	1	3.23%
four one zero seventy five twenty eight	1	3.23%
four ten seventy five twenty eight	1	3.23%
let's try another one	1	3.23%
my account number is three hundred nine five eight one	1	3.23%
nine five eight one	1	3.23%
seven oh two three	1	3.23%
seven zero two three	1	3.23%
two eight five four	1	3.23%
zero	1	3.23%
TOTAL	31	100%

Error

This report provides the transcriptions (in field REF) and natural language references (in field NL REF) for the most common errors—FA-in, FR-in—sorted by frequency of occurrence.

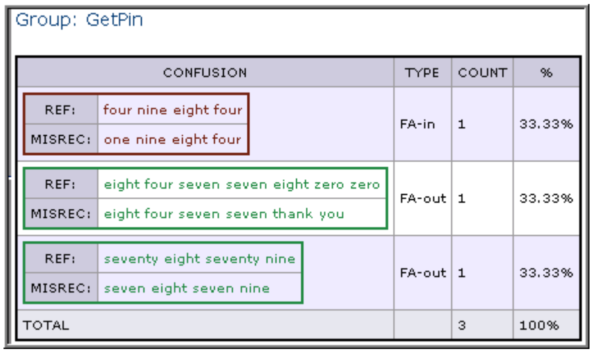


Group: GetBillPayCompany

ERROR	TYPE	COUNT	%
REF: visa NL REF: {<company visa><vws_id 0>}	FA-in	2	33.33%
REF: a t and t NL REF: {<company phone><vws_id 0>}	FR-in	2	33.33%
REF: a t and t NL REF: {<company phone><vws_id 0>}	FA-in	1	16.67%
REF: i don't know NL REF: {<company list><vws_id 0>}	FR-in	1	16.67%
TOTAL		6	100%

Confusion

This reports provides the recognition hypothesis (in field MISREC) and actual transcription (in field REF) for the most common confusions (FA-in, FA-out), sorted by frequency of occurrence.



Group: GetPin

CONFUSION	TYPE	COUNT	%
REF: four nine eight four MISREC: one nine eight four	FA-in	1	33.33%
REF: eight four seven seven eight zero zero MISREC: eight four seven seven thank you	FA-out	1	33.33%
REF: seventy eight seventy nine MISREC: seven eight seven nine	FA-out	1	33.33%
TOTAL		3	100%

Processing XML reports

The reports are output as XML documents. These reports can be converted via Extensible Stylesheet Language Transformations (XSLT)/ Extensible Stylesheet Language Formatting Objects (XSL-FO) into HTML, PDF, RTF, etc. The tools include default stylesheets that let you view the reports as HTML, simply by viewing the XML documents in Internet Explorer v6.0.

The tools look for a stylesheet of a matching name in the default stylesheets directory (*%NUANCE%\tuner\samples\stylesheets*). For example, for the *accuracy.xml* report, the tools look for stylesheet *%NUANCE%\tuner\samples\stylesheets\accuracy.xsl*. If a matching stylesheet is found, the tools copy the stylesheet to the output directory and add an XML processing instruction in the report referencing this stylesheet. For example, report *accuracy.xml* contains the instruction `<?xml-stylesheet href="accuracy.xsl" type="text/xsl"?>`.This processing instruction tells the web browser to render the XML document according to the referenced stylesheet.

The default stylesheets directory is specified with the *TUNER_STYLESHEETS* variable. By default, this variable is set to *%NUANCE%\tuner\samples\stylesheets*, which contains the default stylesheets provided by Nuance.

If you want to convert the reports to another format such as PDF or RTF or to convert them to HTML offline (for example, via scripts), you can use offline XSLT engines such as:

- **Xalan**—See *xml.apache.org/xalan-j/* for more information
- **Saxon**—See *saxon.sourceforge.net/* for more information

  For example, using Saxon, you can transform an XML document as follows:

```
> java -jar saxon.jar -o html\status.html analysis\status.xml
analysis\status.xsl
```

And since *status.xml* itself specifies *status.xsl* as the stylesheet of choice via an XML processing instruction, the above can be simplified to:

```
> java -jar saxon.jar -a -o html\status.html
analysis\status.xml
```

For more information about:

- XML, see *www.w3.org/XML/*
- XSLT and XSL-FO, see *www.w3.org/Style/XSL/*

Working with the Records and Scores reports

The Analysis and Tuning Tools can generate the following two very detailed reports:

- Records report (*records.xml.gz* file): This report provides detailed information from the call log data. For example, it provides the recognition results, call durations, call begin and end times, and so on, for all the call log records in a call. This report does not require transcriptions. It is useful for analyzing data before transcriptions are available, to help identify and investigate potential problem areas flagged by the Status report, and so on. For example, if a group contains a high number of SPEECH_TOO_EARLY utterances, you can use the Records report to listen to these utterances (and the ones surrounding it).
- Scores report (*scores.xml.gz* file): This report contains the raw scores for each transcribed utterance. It is useful when you want to investigate specific problems identified by the standard reports. For example, you might want to listen to all of the confusion utterances to hear callers' pronunciations.

The reports are provided in a gzipped file that can be viewed in a standard text editor by simply gunzipping it.

Understanding the contents of the Records report

Like the other reports, the Records report is an XML file (*records.xml*).

The Records report provides the recognition results for all the call log records in a call. The Records report, which does not require transcriptions, is an unsupervised report, which means that no records are thrown out (as is the case for supervised reports; see “Supervised reports” on page 168).

Under the root element of `<report>`, the data scored for each `<call>` is stored in `<record>` elements. Here is an example of a `<record>` element:

```
<record gdef-instance="7745209"
         group="GetPizzaSize"
         utt="C:\Nuance\V8.5.0/tuner/samples/data/PizzaTalk/
          logs/Nuance/PizzaTalk/2003/02February/20/12/
          09-29-zamboni-001_5060-utt01.wav">
   <fields>
     <field name="STATUS">
       <![CDATA[RECOGNITION]]>
     </field>
   </fields>
   <recresult top-conf="81"
              type="RECOGNITION">
     <result recog="a large pizza">
       <![CDATA[{<size large><vws_id 0>}]]>
     </result>
   </recresult>
 </record>
```

As shown above, the call log `<record>` element contains pointers to the utterance (as seen by the tools). It also references one of the grammar definition instances in the Groups report, through attribute `gdef-instance`, as well as the group name.

The record also contains the `<fields>` element, which contains specific fields from the call log records. By default, only the STATUS `<field>` is displayed to conserve disk space. To include other fields, set the *tuner-config* parameter `FieldsToDump` for each field to include. See “Specifying call log fields to include in Records and Scores reports” on page 156 for more information.

The call log `<record>` element also includes the `<recresult>` as seen in deployment.

You can use the Records report to:

- Generate transcriptions for the utterances. Using the *records_playlist.xsl* stylesheet described below, you can have a first pass at generating transcriptions based on the top hypothesis.
- Create a call-based dataset with all the records specified with the *tuner-config* parameter `BaseLogDir`. Using the *records_dataset.xsl* stylesheet described below, you create a dataset that you can then edit to specify only the records and utterances you want to process when running *nuance-analyze* and *nuance-tune*. See “Filtering data based on a dataset file” on page 145 for more information.

Note: To look at the Document Type Definition (DTD) for the Records file, see "DTD for the records file" on page 231.

To generate the Records report, run the *nuance-analyze* tool with the `-dump_records` option.

Working with the records

Since the Records report is an XML document, you can parse and process the contained information just like you would any other XML data. You can parse the raw data with your own code and compute your own reports from this data. You can also write stylesheets that extract and present information from the Records report as desired.

You can also extract scores data into a comma-separated-value (CSV) file. This format is easy to import into tools like Excel to make use of its many features, such as filtering. For example, with Microsoft Excel 2000 you can quickly filter, organize, and analyze data using the Auto Filtering feature, as follows:

1. Open the CSV file in Microsoft Excel 2000. (Make sure to select 'Comma' as the delimiter when importing the file.)
2. Select Data->Filter->AutoFilter.

   You can then find and work with a subset of your data. The filtered list displays the rows that meet the criteria you specify for a column. See the Microsoft Excel help for more information.

The tools include several sample stylesheets that demonstrate how to work with the Records report:

Note: The Records report stylesheets are located in the *%NUANCE%\tuner\samples\stylesheets* directory.

stylesheet	Description
records_csv.xsl	Extracts most of the raw record information into a CSV file. Note that only the top hypothesis in an N-best list is extracted into the CSV file.
records_transcriptions.xsl	Creates a template transcriptions file from the Records report. The top hypothesis, if any, is used as the initial guess at a transcription.
records_playlist.xsl	Creates a playlist file (usable by *playwav*) from the Records report.
records_dataset.xsl	Creates a call-based dataset (to use with the *-dataset* filter) from the Records report. See "Filtering data based on a dataset file" on page 145.

You can use any of these stylesheets with your favorite XSLT engine. For example, to generate the dataset file from the Records report, you could use Saxon as follows:

```
> java -jar saxon.jar -o dataset.xml records.xml
%NUANCE%\tuner\samples\stylesheets\records_dataset.xsl
```

Understanding the contents of the Scores report

Like the other reports, the Scores report is an XML file (*scores.xml*).

Under the root element of `<report>`, all of the data scored for each `<call>` is listed under a separate `<datum>` element. Here is an example of a `<datum>` element:

Note: To look at the Document Type Definition (DTD) for the scores file, see "DTD for the scores file" on page 228.

```
<datum>
  <record gdef-instance="2876"
        log="/MyData/2001/06June/18/09-25-43-VR35MROD-1_9/LOG"
        utt="/MyData/2001/06June/18/09-25-43-VR35MROD-1_9/utt01.wav">
    <fields>
      <field name="STATUS">
        <![CDATA[RECOGNITION]]>
      </field>
    </fields>
    <recresult top-conf="55"
               type="RECOGNITION">
      <result recog="speech">
        <![CDATA[{<cmd speech>}]]>
      </result>
    </recresult>
  </record>
  <transcription clean="speech"
                 labels="male"
                 orig="speech"/>
  <nl-ref>
    <![CDATA[{<cmd speech>}]]>
  </nl-ref>
  <score acc-stat="CA-in"
         acc-stat-no-rej="CA-in"
         ig-stat="IG"
         top-correct="0"/>
</datum>
```

As shown above, a `<datum>` element contains the call log `<record>` element, which contains pointers to the utterance (as seen by the tools) as well as the containing log file. It also references one of the grammar definition instances in the Groups report, through attribute `gdef-instance`. The record also contains the `<fields>` element, which contains specific fields from the call log records.

By default, only the STATUS `<field>` is displayed to conserve disk space. To include other fields, set the *tuner-config* parameter `FieldsToDump` for each field to include. See "Specifying call log fields to include in Records and Scores reports" on page 156 for more information.

The call log `<record>` element also includes the `<recresult>` as seen in deployment (*nuance-analyze*) or as recomputed (*nuance-tune*).

After the `<record>` element, the `<datum>` element contains the `<transcription>` provided in the input transcriptions files. It provides both the original transcription (with attribute `orig`) as well as the clean transcription (all the markers are removed except for `[fragment]` and `(())`).

The natural language reference (`<nl-ref>`) produced from the clean transcription follows the `<transcription>` element.

Finally, the raw `<score>` itself for the `<datum>` is provided. It contains the following information:

- `acc-stat`—Accuracy label with rejection for the first result in the RecResult: `CA-in`, `FA-in`, `FA-out`, `FR-in`, or `CR-out`
- `acc-stat-no-rej`—Accuracy label with no rejection for the first result in the RecResult: `CA-in`, `FA-in`, `FA-out`, `FR-in`, or `CR-out`
- `ig-stat`—Indicates whether the utterance was in grammar (`IG`), out of grammar but not out of coverage (`OOG`), or out of coverage (`OOC`)
- `top-correct`—Index of which result in the N-best list was CA-in (if any)

Working with the scores

The tools include several sample stylesheets that demonstrate how to work with the Scores report:

Note: The Scores report stylesheets are located in the *%NUANCE%\tuner\samples\stylesheets* directory.

stylesheet	Description
scores_csv.xsl	Extracts most of the raw scores information into a CSV file. Note that only the top hypothesis in an N-best list is extracted into the CSV file.
scores_ig.xsl	Extracts the in-grammar utterances from the Scores report into an HTML file. You can listen to the utterances by clicking on them from your web browser.
scores_ig_csv.xsl	Same as above, but presents the information as a CSV file.

stylesheet	Description
scores_ig_cgp.xsl	Extracts the in-grammar utterances from the Scores report into a format suitable for use by *compute-grammar-probs*. You may need to massage grammar names in the output, since *compute-grammar-probs* only works with static grammars.
scores_oog.xsl	Extracts the out-of-grammar and out-of-coverage utterances from the Scores report into an HTML file. You can listen to the utterances by clicking on them from your web browser.
scores_oog_csv.xsl	Same as above, but presents the information as a CSV file.
scores_error.xsl	Extracts the error (FA-in, FR-in) utterances from the Scores report into an HTML file. You can listen to the utterances by clicking on them from your web browser.
scores_error_csv.xsl	Same as above, but presents the information as a CSV file.
scores_confusion.xsl	Extracts the confusion (FA-in, FA-out) utterances from the Scores report into an HTML file. You can listen to the utterances by clicking on them from your web browser.
scores_confusion_csv.xsl	Same as above, but presents the information as a CSV file.

You can use any of these stylesheets with your favorite XSLT engine. For example, you could use Saxon as follows:

```
> java -jar saxon.jar -o confusion.csv scores.xml
%NUANCE%\tuner\samples\stylesheets\scores_confusion_csv.xsl
```

Chapter 10

Creating application-specific dictionaries

10

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 Chapter 6, “Adding missing pronunciations” on page 107, 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 North American English:

Table 8: Mappings for North 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		
Fricatives	**f**	***f***riend	**v**	***v***oice
	T	pa***th***	**D**	***th***em
	s	***s***it	**z**	***z***ebra

Table 8: Mappings for North American English

Phonetic Category	Phoneme	Example	Phoneme	Example
	S	***sh**ield*	**Z**	vi**s**ion
	h	***h**ave*		
Affricates	**tS**	***ch**ur**ch***	**dZ**	***j**u**dg**e*
Approximants	**j**	***yes***	**w**	***w**in, **wh**ich*
	r	***row***	**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; perhaps providing a variant with each will be 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*

Table 9: Mapping English-language letters

Letter	Phonemes	Example
	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	giant
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*
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***

Table 9: Mapping English-language letters

Letter	Phonemes	Example
	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
t	**t**	***t***ime
	!	bu***tt***er, sani***t***y
	D	***th***em
	T	***th***in
	tS	ques***ti***on
	S	na***ti***on
u	^	***u***nder, p***u***tt
	U	p***u***t, w***oul***d

Table 9: Mapping English-language letters

Letter	Phonemes	Example
	u	l***u***nar, r***u***de
	j u	***u***sage, conf***u***se
	*	foc***u***s
v	**v**	***v***ery
w	**w**	***w***isdom, ***w***hich
x	**z**	***x***enophobia
	k s	a**x**is
y	**j**	***y***es
	i	ver***y***
	aj	cr***y***
z	**z**	***z***ebra

As a general guideline for generating dictionary entries, ignore the spelling and focus on how the word sounds. Break it down into its component phonemes sound by sound. Write the codes for those phonemes in the correct sequence. To verify, try to reproduce the sound of the word by reading the symbols in sequence. For example, in the word *backwards*, the sounds (represented by spelling, not phonemes) are as follows:

b, a, ck, w, ar, d, s

The letters are sufficient to tell you what the component sounds are, only because you can still see what the word is, and you know how to pronounce it. To tell the recognizer that the sounds are spelled b-a-c-k-w-a-r-d-s, you must be more precise about the exact sounds:

- In the Nuance phoneme set, the [b] sound is straightforward, and the phoneme symbol for it is /b/.
- The next sound is a little harder. The letter *a* can represent many different sounds, even in this one word. The English letter table shows that the second sound in *backwards* is the same as the first sound in *apple*, and is represented by the symbol/a/.
- The third sound is represented in the spelling by two letters, *ck*, but the symbol for this phoneme is just /k/.

- The fourth phoneme is represented by the symbol `/w/`.
- The fifth phoneme is tricky. It is a syllabic *r*, meaning that it has blended with the vowel to make up the root of the syllable, and is therefore one sound. The symbol for this phoneme is `/*r/`. (In a Boston accent, the *r* part of the phoneme is dropped, and the phoneme is `/*/`.)
- The symbol for the sixth phone is `/d/`, and the final phone, which is spelled with an *s*, is actually represented by the symbol `/z/`, which is what it sounds like.

The dictionary entry for this word, therefore, might look like this:

```
backwards         b a k w *r d z
backwards         b a k w * d z
```

where the second pronunciation indicates a Boston accent.

Note: Be especially careful with words borrowed from other languages, words that would be hard for a child to read, and all vowels. Also, the letter *s* can correspond to many sounds, such as the phoneme `/z/` as it does in *busy* and *dogs*. Similarly, the letter *d* can correspond to the phoneme `/t/`, as in *asked*, *missed*, and *pushed*.

Multiple pronunciations

You can provide multiple pronunciations for any word. The number of pronunciations you want to provide depends on how many dialects you want your system to recognize. List multiple pronunciations on separate lines. For example, the following table lists some examples of words that could have multiple pronunciations:

Table 10: Examples of words with multiple pronunciations

Word	Phonemes
either	aj D *r
	i D *r
compass	k ^ m p * s
	k A m p * s
defense	d * f E n s
	d i f E n s
sandwich	s a n d w I tS

Table 10: Examples of words with multiple pronunciations

Word	Phonemes
	s a n w I tS

Generating new pronunciations

The best way to generate new pronunciations for long lists of words is to use the utility *words2dict*, which automatically generates pronunciations for missing words. You can then correct them manually.

To use the *words2dict* utility, enter:

```
> words2dict package input_word_list output_dictionary
-dont_use_system_dict
```

For example:

```
> words2dict English.America.3 words.list words.dict
-dont_use_system_dict
```

See the *Nuance API Reference* for more information.

Sample dictionary file

The following lists a sample dictionary made up of the words in Table 9.

```
able         e b * l
adam         a ! * m
agent        e dZ * n t
alive        * l aj v
aphid        e f * d
apple        a p * l
asked        a s k t
ball         b O l
bird         b *r d
book         b U k
bought       b O t
butter       b ^ ! *r
cat          k a t
caught       k O t
child        tS aj l d
coat         k o t
compass      k ^ m p * s
confuse      k * n f j u z
cow          k aw
cry          k r aj
dogs         d A g z
```

dogs	d O g z
echo	E k o
even	i v * n
father	f A D *r
focus	f o k * s
golf	g A l f
grammar	g r a m *r
hotel	h o t E l
india	I n d i *
iraq	* r A k
iraq	I r A k
iraq	aj r A k
iraq	* r a k
iraq	I r a k
iraq	aj r a k
ivory	aj v *r i
ivory	aj v r i
jump	dZ ^ m p
key	k i
kissed	k I s t
knot	n A t
law	l O
lisa	l i s *
loop	l u p
lunar	l u n *r
mike	m aj k
nation	n e S * n
new	n u
new	n j u
olive	A l * v
open	o p * n
pan	p a n
pawn	p O n
proceed	p r * s i d
proceed	p r o s i d
pull	p U l
put	p U t
putt	p ^ t
question	k w E s tS * n
question	k w E S tS * n
quick	k w I k
rider	r aj ! *r
ring	r I g~
rude	r u d
sanity	s a n * ! i
shell	S E l
shield	S i l d
shoe	S u

```
show        S o
soup        s u p
them        D E m
thin        T I n
thumb       T ^ m
time        t aj m
under       ^ n d *r
usage       j u s * dZ
usage       j u s I dZ
very        v E r i
vision      v I Z * n
which       w I tS
widen       w aj ! * n
wisdom      w I z d * m
would       w U d
xenophobia  z i n * f o b i *
yes         j E s
zebra       z i b r *
```

To create pronunciations, you can also use the command-line program *pronounce* to get pronunciations for words that have similar pronunciations. For example, to generate a pronunciation for the word “glyph,” you could base it on the pronunciation for the word “cliff.” To use *pronounce*, specify the master package you are using and a list of one or more words for which you would like to see pronunciations. See the online documentation for *pronounce* for more details.

Using an application-specific dictionary

To use an application-specific dictionary:

1 Create the dictionary file as specified in this appendix.

2 Recompile your package using the *-merge_dictionary* or *-override_dictionary* option, as appropriate.

- For more information about the *-merge_dictionary* and *-override_dictionary* options, see “Merging and overriding dictionaries” on page 108.
- For more information about recompiling your package, see “Summary of compiler options” on page 97.

3 If you precompile your grammars into NGOs, use the same dictionary options you specified when recompiling your package. For example:

- If you used `-merge_dictionary myMergeDict` when compiling your package, specify `comp.MergeDictionary=myMergeDict` when running *nuance-compile-ngo*.

- If you used `-override_dictionary myOverrideDict` when compiling your package, specify `-dict myOverrideDict` when running *nuance-compile-ngo*. (Note that using `-dict myOverrideDict` is the same as using `comp.OverrideDictionary=myOverrideDict`)

See “Dynamic compilation to a file” on page 98 for more information about *nuance-compile-ngo*.

Converting non-CPA dictionaries

If you have Nuance dictionary files you created with a version of the Nuance System older than 6.2, you need to convert these to CPA. The Nuance System includes the tool *arpabet-to-cpa* that can do this for you. See the online documentation for usage information.

Phoneme sets for other languages

In addition to North American English, Nuance supports a variety of other languages and English dialects. To work with other languages, you need to use a specific Nuance master package to process that language. See the section “Choosing a master package” on page 103.

Tables for mapping the phonemes for all supported languages to their CPA representation are located at the documentation site on Nuance Technical Support Online at *support.nuance.com*.

Appendix A

GSL reference

A

This appendix provides a formal definition of the syntax of the Grammar Specification Language (GSL), and a summary of all the package files used to compile a package.

The following conventions are used in the syntax description listed below:

Expression	Meaning
X	A nonterminal
“xyz”	Literal string xyz should be typed into the specification
X ::= Y	X consists of Y
X | Y | Z	X or Y or Z
X Y	X followed by Y, with white space allowed
X~Y	X followed by Y with no white space intervening

GSL syntax

The following rules formally define the GSL syntax. However, these rules may—in a few cases—generate some non-valid expressions, and therefore, their use should be complemented with the notes following the syntax rules.

```
GrammarFile ::= GrammarDefinitions
GrammarDefinitions ::=
      GrammarDefinition |
      GrammarDefinition GrammarDefinitions
GrammarDefinition ::=
      GrammarNameWithOptionalSuffixes GrammarConstruct |
      GrammarNameWithOptionalSuffixes
```

```
GrammarNameWithOptionalSuffixes ::=
      GrammarName~":"~Suffixes |
      GrammarName
Suffixes ::= Suffix | Suffix~","~Suffixes
Suffix ::= "dynaref" | "dynamic" | "filler"
GrammarConstruct ::= BareConstruct |
      BareConstruct Commands |
      "=" GrammarConstruct |
      GrammarName~":slm" "\"AnyString "\" |
      GrammarName~":slm" "\"AnyString "\" "="
GrammarConstruct
BareConstruct ::=
      BareConstructWithoutProb    |
      BareConstructWithoutProb~"~"~Prob
BareConstructWithoutProb ::= Word      |
      GrammarReference |
      "?" GrammarConstruct                |
      "*" GrammarConstruct                |
      "+" GrammarConstruct    |
      "[" GrammarConstructs "]" |
      "(" GrammarConstructs ")"
GrammarConstructs ::=
      GrammarConstruct |
      GrammarConstruct GrammarConstructs
Word ::= LowerCaseLetter | Digit | WordChar~Word
GrammarReference ::=   GrammarName |
      GrammarName~":"~Variable |
      Ext_Grammar_Ref
GrammarName ::= LimitedString
Commands ::= "{" CommandList "}"
CommandList ::= Command | Command CommandList
Command ::= "<" Slot Value ">" |
      "return (" Value ")"  |
        "insert-begin (" Variable Value ")" |
        "insert-end (" Variable Value ")" |
      "concat (" Variable Value ")"
Slot ::= SlotString
SlotString ::= VOrFChar | VOrFChar~SlotString
Values ::= Value | Value Values
Value ::= Integer |
      LimitedString |
      """~AnyString~""" |
      Structure   |
      "$"~VariableExpression |
      List |
      FunctionApplication
Structure ::= "[" FeatureValuePairs "]"
```

```
FeatureValuePairs ::=
      FeatureValuePair |
      FeatureValuePair FeatureValuePairs
FeatureValuePair ::= "<" Feature Value ">"
Feature ::= VariableOrFeatureString
Variable ::= VariableOrFeatureString
VariableExpression ::= Variable | Variable~"."~FeaturePath
FeaturePath ::= Feature | Feature~"."~FeaturePath
List ::= "()" | "(" Values ")"
FunctionApplication ::= Function~"(" Values ")"
Function ::= LimitedString
WordChar ::= LowerCaseLetter | "-" | "_" | "." | "@" | "’" | Digit
LimitedString ::= LimitedChar | LimitedChar~LimitedString
LimitedChar ::= UpperCaseLetter |
      LowerCaseLetter |
      Digit    |
      "-" | "_" | "@" | "’" | "."
Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
VariableOrFeatureString ::=
      VOrFChar |
      VOrFChar~VariableOrFeatureString
VOrFChar ::= UpperCaseLetter |
      LowerCaseLetter |
      "-" | "_" | "@" | "’"
AnyString ::= Char | Char~AnyString
```

Notes

- A semicolon indicates a comment. The compiler ignores everything on the line, after a semicolon.
- Grammar names must contain at least one uppercase letter.
- White space is a sequence of at least one of the following characters: space, tab, new line, return, and page feed.
- `Char` signifies any character other than the double quote or white space.
- `Prob` denotes a *non negative* floating point number not greater than 1.0.
- The following strings are reserved for internal use and cannot be used to denote a word or grammar name (*n* designates any integer):

  AND-*n*, OR-*n*, OP-*n*, KC-*n*, PC-*n*

External grammar reference syntax

The Grammar Specification Language has been extended to include the following grammar reference syntax rules.

```
Ext_Rule_ref ::= ExRef_Resolved | "<" ExRef_Resolved "%" ExRef_Special ">"
```

```
ExRef_Resolved ::= "<" ExRef_Simple
                 ["!" DynGrammLabel "=" ExRef_Simple
                    ["&" DynGrammLabel "=" ExRef_Simple]*] ">"

ExRef_Simple ::= "<"ExRef_Type~":"~ExRef_Path[~"#"~ExRef_StartRule]">"
ExRef_Type ::= "http" | "file" | "dgdb" | "static" | "builtin"
ExRef_Special ::= "<special:"~Special_Path~">" | "<fail:"~ExRef_Path~">"
Special_path ::= "NULL" | "VOID" | "passthrough" | "roadblock" |
              "resistor?weight="~Prob
```

Some notes:

- The `Special_Path` specification `NULL` is equivalent to `passthrough`, and the `Special_Path` specification `VOID` is equivalent to `roadblock`.
- `ExRef_Path` is any string not containing the characters # or >.
- `ExRef_StartRule` is any valid public rule name.

`DynGrammLabel` is the name of a grammar annotated as `:dynamic []` in the associated simple reference.

Summary of package files

The table below lists all the files that *nuance-compile* takes into account when compiling a recognition package, with a brief description of the expected contents.

The common name of these files is the name of the package being compiled. For example, you compile the file *myapp.grammar* to generate a recognition package called *myapp*. All files should be stored in the same directory.

Table 1: Recognition package files

File extension	Required?	Contents
.grammar	Yes	GSL code
.dictionary	No	One word (or compound) followed by a sequence of phonemes per line
.slot_definitions	No	One slot name per line
.functions	No	User-defined functions

Advanced SLM features

B

This section provides advanced information on the Nuance SLM utilities. If you are new to SLM features, please follow the procedures described in Chapter 5, “Say Anything: Statistical language models and robust interpretation”.

For background information on SLMs, please see the following books:

- F. Jelinek, *Statistical Methods for Speech Recognition*, The MIT Press, Cambridge, 1997.
- S. Young, and G. Bloothooft, *Corpus-Based Methods in Language and Speech Processing*, Kluwer Academic Publishers, Norwell, 1997.

The *train-slm* tool

The *train-slm* tool, Nuance's SLM trainer, is a command-line tool that trains n-gram language models.

Training an n-gram model is done in four phases:

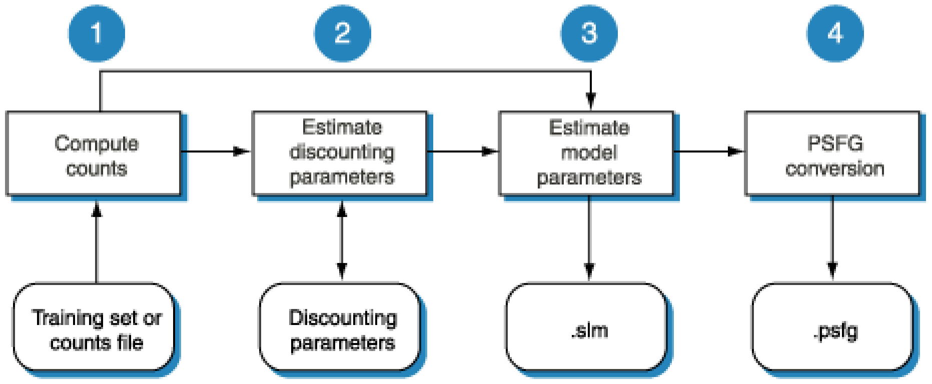
1 Load a training set, either from a text file (one sentence per line) or from a counts file. This phase computes the counts of all n-grams that occur in the corpus.

2 Estimate the discounting parameters that will be used to smooth the relative frequencies.

3 Estimate the model probabilities from the counts.

4 Convert the estimated model to a Probabilistic Finite State Grammar (PFSG), the structure used to compile recognition packages.

This process is illustrated in the figure below.


1
2
3
4
Compute counts
Estimate discounting parameters
Estimate model parameters
PSFG conversion
Training set or counts file
Discounting parameters
.slm
.psfg

The trainer can perform the four steps in a single command-line invocation, as described in Chapter 5, or you can invoke the trainer to perform individual tasks. You can save the output of the intermediate steps to files and use them in future runs of the trainer. This capability lets you, for example, run step 3 again with different parameters without having to repeat step 1 and step 2.

Once the model is trained, it can be saved in SLM format in a *.slm* file and in PFSG format in a *.pfsg* file. The SLM format is a proprietary format for storing the language model. It is the input to *process-slm*, the utility that performs post-training operations such as perplexity measurements and conversion to PFSG. The PFSG format is the result of converting a language model to a Probabilistic Finite State Grammar, the data structure used by the Voyager recognizer. This is the file read by *nuance-compile* to produce recognition packages.

Usage

The *train-slm* tool can be used with either a text corpus or a counts file as input:

```
train-slm
   -corpus input-corpus-filename | -counts input-counts-filename
      [DISCOUNTING_OPTIONS]
      [SLM_OPTIONS]
      [OPTIONS]
```

where `[DISCOUNTING_OPTIONS]` are:

```
   [   -discounts input-discounts-filename |
      -discounting-type   GOOD-TURING |
                          ABSOLUTE_DISCOUNTING |
                          KATZ_DISCOUNTING
      [-discounting-observ num-observ] ]
```

where `[SLM_OPTIONS]` are:

```
[   -slm output-slm-basename
   [-pfsg-type   STANDARD [-keep-lowprobs] |
                 FULL-EXPANSION |
                 ADJUST-BOW ]
   [-no-pfsg]]
```

where `[OPTIONS]` are:

```
[-help]
[-no-unknown]
[-n N]
[-vocab vocab-filename]
[-verbose verbose_level]
[-write-counts output-counts-filename]
[-write-discounts output-discounts-filename]
```

Required input

The *train-slm* tool requires one of the following input files, specified with the *corpus* or *counts* option:

`-corpus input-corpus-filename`

(step 1) Loads the n-gram counts from a text corpus file. The corpus files contains one sentence per line. See "Create the training set" on page 75 for more information.

`-counts input-counts-file`

(step 1) Loads the n-gram counts from a counts file. Each line of the counts file contains an n-gram (a sequence of n words) followed by an integer (the count for that n-gram).

Discounting options

The *train-slm* tool can take the following optional discounting options:

```
[   -discounts input-discounts-filename |
   -discounting-type     GOOD-TURING |
                         ABSOLUTE_DISCOUNTING |
                         KATZ_DISCOUNTING
   [-discounting-observ num-observ] ]
```

`-discounts input-discounts-filename`

(step 2) Instead of estimating the discounting parameters from the loaded corpus, reads them from file *input-discounts-filename*. Typically, you produce this file by running *train-slm* with the *-write-discounts* option. You can then edit it manually to change the discounting parameters. See "Editing discounting parameters" on page 204 for more information.

`-discounting-type discounting-method`

(step 2) Controls which type of discounting algorithm gets trained. This option cannot be used with the `-discounts` option: the discounting parameters are either loaded from a file or estimated. Three discounting methods are supported: Each method uses discounting parameters that are estimated from the loaded corpus.

- `GOOD-TURING` (default)—The Good-Turing formula is used for counts in the range [`lower_bound`, `upper_bound`] where `lower_bound` and `upper_bound` are parameters estimated by *train-slm*. Under `lower_bound`, counts are discounted to 0. Above `upper_bound`, counts are multiplied by a discounting factor `alpha`, a number close to but smaller than 1.0.
- `ABSOLUTE_DISCOUNTING`—A count of one is reduced by a constant `b1`, estimated by *train-slm*. All other counts are reduced by the same constant, `b2`. The algorithm also uses lower and upper bounds for discounting. Below the `lower_bound`, a count is discounted to 0; above the `upper_bound`, no discounting occurs.
- `KATZ_DISCOUNTING`—This is a variant of `GOOD-TURING` in which the counts in [`lower_bound`, `upper_bound`] are discounted by using the Good-Turing formula and multiplied by a discounting factor `alpha` that is typically close to 1.0. Above `upper_bound`, the counts are unchanged.

`-discounting-observ` *`num-observ`*

(step 2) Controls the number of observations used to estimate the discounting parameters. Only the first *`num-observ`* observations from the training corpus are used to estimate the parameters. Because the vocabulary must not be applied to the training corpus to train the discounting methods, step 2 can take a lot of memory, even more than the model training itself. This option is useful to limit the amount of memory used by step 2. The default behavior is to use the full training set. Note that this option cannot be used with the *-discounts* or *-counts* options.

SLM options

The *train-slm* tool can take the following SLM options:

```
[   -slm output-slm-basename
   [-pfsg-type   STANDARD [-keep-lowprobs] |
                 FULL-EXPANSION |
                 ADJUST-BOW ]
   [-no-pfsg] ]
```

`-slm` *`output-slm-basename`*

(step 3) Tells the trainer to train an SLM. The resulting model is saved to file *`output-slm-basename`.slm* in SLM format and to *`output-slm-basename`.pfsg* in PFSG format.

`-pfsg-type` *`pfsg-type`*

(step 4) Controls the conversion algorithm used to produce the PFSG. Three methods are available:

- `STANDARD` (default)—Creates a standard PFSG where a backoff distribution is used for n-grams that were not observed in the training set. Low probability n-grams are pruned (see `-keep-lowprobs`).
- `FULL-EXPANSION`—The backoff distributions are not used and the low probability n-grams are NOT pruned. Each distribution has V transitions, where V is the vocabulary size.
- `ADJUST-BOW`—Similar to the `STANDARD` conversion but the backoff weight is decreased to ensure that no low probabilities remain. Thus, low probabilities are NOT pruned.

`-keep-lowprobs`

(step 4) Only valid when *`pfsg-type`* is `STANDARD`. Instructs the trainer not to prune low probabilities when converting to PFSG. A low probability occurs when a distribution word's probability is smaller than the path through the backoff distribution. By default, the low probability n-grams are pruned.

`-no-pfsg`

(step 4) Does not generate the PFSG. By default, a `STANDARD` PFSG is generated.

Additional options

The *train-slm* tool can take the following additional options:

```
[-help]
[-no-unknown]
[-n N]
[-vocab vocab-filename]
[-verbose verbose_level]
[-write-counts output-counts-filename]
[-write-discounts output-discounts-filename]
```

`-help`

Provides a detailed command-line usage.

`-no-unknown`

(Steps 1 and 3) Trains a closed language model, that is, one that does not assign probabilities to phrases containing out-of-vocabulary words. For an open language model, the PFSG uses the UNK grammar rule that must be defined by the user. Closed model training ignores the n-grams whose last word is out-of-vocabulary; also, n-grams whose history contains an out-of-vocabulary word are truncated to erase the word.

The default is to train an open language model and use the unknown class UNK to represent out-of-vocabulary words. See “Open and closed vocabulary SLMs” on page 80 for more information.

`-n` *N*

(Steps 1, 2, and 3) Trains an n-gram SLM of order N. This number determines the order of counts to compute and has implications as to the amount of memory used by *train-slm*. Default is 3 (trigram). See “Determine the order of the model” on page 81 for more information.

`-vocab` *`input-vocab-filename`*

(Steps 1 and 3) Uses the vocabulary file *input-vocab-filename* to train the n-gram SLM. This file contains the words that need to be covered by the produced SLM. See “Create a vocabulary file (optional)” on page 79 for more information.
A vocabulary file contains one word per line. Out-of-vocabulary words are processed depending on whether the option *-no-unknown* is specified or not. Using a vocabulary decreases the memory requirements of the training process.

`-verbose` *`verbose-level`*

(Steps 1, 2, and 3) Controls the amount of information output by the trainer; *`verbose-level`* is an integer between 0 and 4:

- 0—No verbose
- 1—Low verbose
- 2—Medium verbose
- 3—High verbose
- 4—Extreme verbose

`-write-counts` *`output-counts-filename`*

(step 1) Saves the counts that were computed from a corpus to the file specified with *`output-counts-filename`*.

`-write-discounts` *`output-discounts-filename`*

(step 2) Saves the discounting parameters to the file *`output-discounts-filename`*. The saved discounting parameters can be used in a future trainer run with the *-discounts* option.

Editing discounting parameters

You can edit the discounting parameters by creating a parameter file, editing it, and then using *train-slm*, as described below.

1 To output the discounting parameters to a file, enter:

```
> train-slm -corpus input-file -write-discounts
output-discounts-filename other_appropriate_options
```

where `input-file` is the name of the training set, `output-discounts-filename` is the name of the file that will contain the discounting parameters, and `other_appropriate_options` is any of the options you want to specify (such as *-n*). For example:

```
> train-slm -corpus training.txt -write-discounts params.txt
```

This command outputs a text file that contains the discounting parameters according to the discounting type specified: `GOOD-TURING` (default), `ABSOLUTE_DISCOUNTING`, or `KATZ_DISCOUNTING`. See "Discounting options" on page 201 for more information about the discounting parameters.

2 Edit the file as required.

3 To train the SLM grammar using the updated parameters, enter:

```
> train-slm -corpus input-file -discounts input-discounts-filename
other_appropriate_options
```

where `input-file` is the name of the training set, `input-discounts-filename` is the name of the discounting parameter file that you updated, and `other_appropriate_options` is any of the options you want to specify (such as *-n*). For example:

```
> train-slm -corpus training.txt -discounts params.txt
```

The *process-slm* tool

You use *process-slm* to perform post-training operations on SLMs. It takes a *.slm* file produced by *train-slm* or an ARPA file as input and can perform the following operations on it:

- Perplexity measurements
- Conversion to *.pfsg* format
- Model interpolation
- N-gram pruning

The resulting language model is then saved to a *.slm* or a *.pfsg* file.

Usage

The *process-slm* tool has the following command-line syntax:

```
process-slm
   slm_basename1 slm_basename2 ...
   [-ppl-corpus input-corpus-filename]
   [-interpolate interpolated-slm-basename -w wgt1 -w wgt2 ...]
   [-verbose verbose-level]
   [-prune-lowprobs]
   [-pfsg
      [-pfsg-type    STANDARD [-keep-lowprobs] |
                     FULL-EXPANSION |
                       ADJUST-BOW ] ]
```

Required input

`slm_basename1 slm_basename2 ...`

Specifies the path of the models to operate on. You do not need to specify the *.slm* extension. You must specify at least two input models for the *-interpolate* option. For all other options, each model is processed one by one.

Options

`-ppl-corpus` *`input-corpus-filename`*

Computes the perplexity of each input model on the training corpus specified by *`input-corpus-filename`*. The text corpus contains one sentence per line. See "Create the training set" on page 75 for more information.

`-interpolate` *`interpolated-slm-basename`*

Performs the linear interpolation of the input models and saves the result to file *interpolated-slm-basename.slm*. The interpolation weights are specified by the *-w* option. There must be as many *-w* options as there are input models. The weights do not need to sum to one. If they don't, they are normalized.

`-w` *`wgtx`*

Specifies an interpolation weight (a float).

`-verbose` *`verbose-level`*

Controls the amount of information output by the trainer; *`verbose-level`* is an integer between 0 and 4:

- 0—No verbose
- 1—Low verbose
- 2—Medium verbose
- 3—High verbose
- 4—Extreme verbose

`-prune-lowprobs`

Prunes the low probability n-grams from each input model and saves it in SLM format to file *slm_basenameX.pruned.slm*. See the `-keep-lowprobs` option of *train-slm* for more information.

`-pfsg`

Converts each input SLM to a PFSG and saves it to file *slm_basenameX.pfsg*.

`-pfsg-type` *`pfsg-type`*

(step 4) Controls the conversion algorithm used to produce the PFSG. Three methods are available:

- `STANDARD` (default)—Creates a standard PFSG where a backoff distribution is used for n-grams that were not observed in the training set. Low probability n-grams are pruned (see `-keep-lowprobs`).
- `FULL-EXPANSION`—The backoff distributions are not used and the low probability n-grams are NOT pruned. Each distribution has V transitions, where V is the vocabulary size.
- `ADJUST-BOW`—Similar to the `STANDARD` conversion but the backoff weight is decreased to ensure that no low probabilities remain. Thus, low probabilities are NOT pruned.

`-keep-lowprobs`

Only valid when *`pfsg-type`* is `STANDARD`. Instructs the trainer not to prune low probabilities when converting to PFSG. A low probability occurs when a distribution word's probability is smaller than the path through the backoff distribution. By default, the low probability n-grams are pruned.

Appendix C

W3C grammar support

C

The World Wide Web Consortium (W3C) has proposed a syntax for speech recognition grammar formats (SRGF). This proposal includes two syntax forms: Augmented BNF (ABNF) and XML (GrXML).

Nuance supports W3C grammars in the XML format. This section first describes how you can use a W3C grammar with the Nuance System. It then lists the Nuance extensions to the W3C format. Finally, it lists the W3C features not currently supported by Nuance.

Note: This chapter does not explain how to write a W3C grammar. For details on the W3C grammar syntax in GrXML form, see the specification published at *www.w3.org/TR/2001/WD-speech-grammar-20010820/*. Nuance is currently working from the draft dated **August 20, 2001**. Since the W3C specification will likely change, grammars developed in Nuance 8.5 may not be compatible with future releases.

Working with W3C grammars in the Nuance System

Most of the concepts in the Nuance Grammar Specification Language, introduced in this document, are also available in the W3C grammar specification. However, each language has some features and concepts that are slightly different from the other.

This section presents some aspects of W3C grammar features that you should be aware of when using this format with the Nuance System, such as:

- Supported encodings
- Differences in the notion of *token*
- `GARBAGE` rule handling

- DTMF mode

Supported encodings

Nuance supports the following encodings in a GrXML specification:

- UTF-8
- UTF-16
- ISO-8859-1
- US-ASCII

Support for other encodings may be added in future releases.

Note: You must make sure that the locale setting for your environment is appropriately set for the language of the grammar you're compiling. See the *Nuance System Installation Guide* for more information.

Tokens

Unlike the GSL specification syntax for a word, a token in a W3C grammar may contain whitespace—if properly quoted—and may contain uppercase characters.

To reconcile this distinction, any token in a W3C grammar that contains whitespace is further broken into individual words by the Nuance System. The Nuance multi-word feature—which allows for pronunciation of a group of words, such as “what is” and “new york”—works at dictionary lookup time and performs its search across *separate* words composing the multi-word. For more information, see “Phrase pronunciations” on page 110.

W3C grammars allow for capital letters in a token. However, for dictionary lookups and the auto-pronunciation to work, the Nuance implementation converts all letters to lowercase as part of the compilation process. This means that you cannot provide separate pronunciations for the same word written with different cases.

Handling of the GARBAGE rule

Nuance maps the special `GARBAGE` rule definition:

```
<ruleref special="GARBAGE"/>
```

to the following block:

```
<item repeat=”0-”>
   @reject@
```

```
</item>
```

This amounts to 0 or more occurrences of the `@reject@` word being recognized wherever the `GARBAGE` ruleref is used.

Nuance recommends that you tune the value of the recognition parameter `rec.RejectWeight` when using the `GARBAGE` rule in your grammars.

Caution: To minimize inaccuracies that the `GARBAGE` rule processing may produce, use this special rule with care. Instead, try to come up with an explicit model of the expected words—or utterances—that you want your grammar to discard. For example, you could train these models using Say Anything grammars. See Chapter 5, "Say Anything: Statistical language models and robust interpretation," for more information.

DTMF mode

The W3C specification indicates that a grammar can be in one of two modes: `voice` (the default mode) or `dtmf`, depending on the type of input that the speech engine should be detecting.

As with GSL, you can specify DTMF sequences in a W3C `voice` grammar, using the tokens dtmf-1, dtmf-2, and so on. These tokens are only valid in a `voice` grammar. In a `dtmf` grammar, an automatic translation of phone buttons to dtmf tokens takes place, according to the following map:

Phone button	dtmf token
0	dtmf-0
1	dtmf-1
2	dtmf-2
3	dtmf-3
4	dtmf-4
5	dtmf-5
6	dtmf-6
7	dtmf-7
8	dtmf-8
9	dtmf-9
star	dtmf-star

Phone button	dtmf token
*	dtmf-star
pound	dtmf-pound
#	dtmf-pound

Any token outside this range will cause a compilation error. Note that, even though a grammar may be in `dtmf` mode, it will still be processed in parallel with the recognition engine, natural language will still be interpreted by a recognition server, and high background noise may cause a rejection.

Note: When the Nuance System handles the telephony, DTMF-only recognition—that is, with no speech—is not supported.

Nuance extensions to W3C grammars

Nuance has extended the GrXML grammar specification to provide additional functionality available in GSL grammars.

The standard GrXML elements that were extended include:

- `<rule>`
- `<ruleref>`

The Nuance-specific GrXML attributes and elements are all in the namespace `http://voicexml.nuance.com/grammars`, as shown in the following DTD specification fragment:

```
<!ATTLIST grammar
  ...
  <!-- Nuance extensions -->
  xmlns:nuance CDATA #FIXED "http://voicexml.nuance.com/grammar"
  ...>
```

The rest of this document will use `nuance:` as an abbreviation for this namespace.

The <rule> element

The `<rule>` element was extended to include the `nuance:dynamic` attribute; this attribute lets you declare a grammar rule as dynamic.

The extension to the DTD that defines this attribute is (extension in bold face):

```
<!ELEMENT rule (%rule-expansion; | example)*>
```

```
<!ATTLIST rule
   ...
   nuance:dynamic (true | false) “false”
   ...>
```

Here is an example of how to declare a rule to be dynamic:

```
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:nuance="http://voicexml.nuance.com/grammar"
         xml:lang="en" version="1.0">
   <rule id="personal" nuance:dynamic="true">
    ...
   </rule>
</grammar>
```

To see more examples of dynamic rules, see “Sample grammars” on page 216.

The <ruleref> element

The `<ruleref>` element was extended to:

- Allow for the redefinition of external rules from the reference.
- Specify a backoff reference in case the referenced rule cannot be resolved—for example, because of an invalid URL or a timeout.

Note: The just-in-time grammar specification provides two operators to handle these situations: the operator `!`, to specify the redefinition of external rules, and the operator `%`, to specify backoff references. See “External rule references and redefinitions” on page 27.

The Nuance extension to `<ruleref>` allows it to contain, in addition to the W3C-specified elements:

- Zero or more `<nuance:redef>` elements
- An optional `<nuance:backoff>` element

Note that only an *external* reference `<ruleref>`, specified with the attribute `uri`, may contain redefinitions and backoffs.

The portion of the expanded DTD that accounts for these elements is:

```
<!ELEMENT nuance:redef EMPTY>
<!ATTLIST nuance:redef
   name CDATA #REQUIRED
   uri CDATA #IMPLIED
   special CDATA #IMPLIED>

<!ELEMENT nuance:backoff EMPTY>
```

```
<!ATTLIST nuance:backoff
   uri CDATA #IMPLIED
   special CDATA #IMPLIED>

<!ELEMENT ruleref (nuance:redef*, nuance:backoff?)>
```

For the `<nuance:redef>` element, the `name` attribute specifies the rule in the reference that is being redefined, and the `uri` or `special` attribute specifies the new rule definition.

The optional `<nuance:backoff>` element is used if the original reference cannot be resolved. Here is a GrXML fragment illustrating the use of these elements:

```
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:nuance="http://voicexml.nuance.com/grammar"
         xml:lang="en" version="1.0">
...
   <ruleref uri="http://faraway-server.com/cgi/grammar.pl">
      <nuance:redef name="personal"
             uri="http://myserver/grammars/addrbook.grxml"/>
      <nuance:redef name="gate" special="NULL"/>
      <nuance:backoff special="VOID"/>
   </ruleref>
...
</grammar>
```

To see more examples of how to use the `<ruleref>` element, see “Sample grammars” on page 216.

Unsupported features

This section discusses several W3C grammar features not currently supported by the Nuance System.

Note: Support for some of these features may be added in the near future, as part of option packs or service packs, as appropriate.

ABNF

ABNF, a compact GSL-like optional format, is not supported in this release.

Elements and attributes

The following are currently ignored by the Nuance System:

- `<meta>` elements of type `http-equiv`

- `<lexicon>` elements
- `type` attribute of the `<ruleref>` and `<alias>` elements

The Nuance System generates warnings when these elements are used.

W3C semantic interpretation tags

The exact format for a semantic interpretation tag—natural language interpretation, in GSL terms—in a W3C grammar is not yet specified.

To append a semantic capability to your W3C grammars:

- Use the Nuance natural language interpretation syntax.
- Set the optional grammar element attribute `tag-format` to `Nuance` to specify the semantic format that your grammar is using.
- Do not enclose the semantic tag in curly brackets. In a GrXML document, this semantic tag is already delimited by the tag element, as in the following fragment:

  ```
  <tag><![CDATA[<command $n> <num $n>]]></tag>
  ```

NL features

Some NL operations available in GSL have no counterpart in W3C grammars. Most notably, there is no equivalent to the following way of referencing a grammar in GSL and assigning its returned value to a variable:

```
SomeGrammarName:aVariableName
```

Grammar syntax and semantic tags are completely disjoint in W3C grammars. Thus, the Nuance NL interpretation syntax was expanded to account for this fact by providing the *$return* variable and the *assign* command. See “Assigning variables” on page 56 for more information.

Language specification

W3C grammars may contain language specifications at the grammar, expansion, and individual token level. The Nuance compiler ignores these specifications and tries to generate pronunciations for missing words as best as possible, according to the current master package.

Use the `Nuance-Package-Name` and `Nuance-Config-Name` keys of the `RECOGNIZE` header to select the appropriate master package and compilation server for your language. See “Specifying just-in-time grammar options: Using the keyword RECOGNIZE” on page 42 for more information.

Using W3C grammars

Once your W3C grammar is written, you can use it just as you use any other dynamic grammar, including:

- Through an external rule reference
- In a just-in-time request
- By compiling the grammar to an NGO file
- Using `newDynamicGrammar` functions

Note: The *nuance-compile* utility does not handle the static compilation of a W3C grammar.

See Chapter 3, "Dynamic grammars: Just-in-time grammars and external rule references," for more information on external rule references and just-in-time requests. See Chapter 6, "Compiling grammars," for more information on NGO files. See the *Nuance Application Developer's Guide* for more information about the `newDynamicGrammar` functions.

Sample grammars

The following samples illustrate how to use W3C grammars.

Sample dialer application grammar

This grammar allows the caller to dial people by name or by number, in response to a question such as "who would you like to call?"

Entries dialed by name may either be from a pre-compiled static corporate grammar or a dynamically specified personal grammar. The top-level grammar 'dial' will fill either the number or the name slots. If the name slot is filled, it will fill the source slot to specify whether or not the corporate or personal grammar was used.

```
<?xml version="1.0"?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:nuance="http://voicexml.nuance.com/grammar"
         xml:lang="en" version="1.0" mode="voice"
         tag-format="Nuance" root="dial">

<!-- Alias a standard telephone number grammar that returns -->
<!-- the value of the telephone number recognized. -->
<alias name="number" uri="telephonenumber.gsl" />

<!-- Primary rule -->
<rule id="dial" scope="public">
  <!-- Low-prob the optional pre_hesitation grammar -->
  <item repeat="0-1" repeat-prob="0.01">
    <ruleref uri="#pre_hesitation"/>
  </item>

  <!-- Low-prob the optional pre_filler grammar -->
  <item repeat="0-1" repeat-prob="0.01">
    <ruleref uri="#pre_filler"/>
  </item>

  <!-- The real contents of the grammar -->
  <ruleref uri="#core"/>

  <!-- Low-prob the optional post_filler grammar -->
  <item repeat="0-1" repeat-prob="0.00001">
    <ruleref uri="#post_filler"/>
  </item>
</rule>

<!-- Main contents of the grammar -->
<rule id="core">
  <one-of>
    <item>
      <ruleref alias="number#MAIN"/>
      <tag><![CDATA[<number $return>]]></tag>
    </item>
    <item>
      <!-- if the package into which this is inserted does -->
      <!-- not have a static grammar called CORPORATE, -->
     <!-- backoff to a roadblock -->
      <ruleref uri="static:CORPORATE">
        <nuance:backoff special="VOID"/>
      </ruleref>
      <tag><![CDATA[<source corporate> <name $return>]]></tag>
    </item>
    <item>
      <ruleref uri="#personal"/>
      <tag><![CDATA[<source personal> <name $return>]]></tag>
    </item>
  </one-of>
</rule>

<!-- Dynamic placeholder for the personal grammar -->
<rule id="personal" nuance:dynamic="true">
  <ruleref special="VOID"/>
</rule>

<!-- Fillers -->
<rule id="pre_hesitation">
```

```
  <one-of>
    <item>uh</item>
    <item>um</item>
    <item>hm</item>
  </one-of>
</rule>

<rule id="pre_filler">
  <one-of>
    <item>
      <one-of>
        <item>i would like</item>
        <item>i'd like</item>
        <item>i want</item>
        <item>on</item>
      </one-of>
      to
      <one-of>
        <item>call</item>
        <item>dial</item>
      </one-of>
    </item>
    <item>
      <item repeat="0-1">
        please
      </item>
      call
    </item>
    <item>
      <item repeat="0-1">
        please
      </item>
      dial
    </item>
  </one-of>
</rule>

<rule id="post_filler">
  <one-of>
    <item>please</item>
  </one-of>
</rule>

</grammar>
```

You could then use this grammar in a just-in-time grammar and redefine the personal rule, as follows:

```
<?xml encoding="1.0"?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
                xmlns:nuance="http://voicexml.nuance.com/grammar"
                version="1.0" mode="voice" xml:lang="en-US"
                tag-format="Nuance" root="main">

<rule id="main" scope="public">
  <ruleref uri="dialer.grxml">
    <nuance:redef name="personal" uri="#names"/>
  </ruleref>
</rule>

<rule id="names" scope="public">
  <one-of>
    <item>mom</item>
    <item>dad</item>
  </one-of>
  <tag>return($string)</tag>
</rule>

</grammar>
```

Sample retail application grammar

This example shows a sample retail application grammar for purchasing items in a hockey equipment store:

```
<?xml version="1.0" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:nuance="http://voicexml.nuance.com/grammar"
         xml:lang="en-US" tag-format="Nuance" root="main">

<rule id="main" scope="public">
  <item>
    <item repeat="0-1">
      <ruleref uri="#number"/>
      <tag>assign(n $return)</tag>
    </item>
    <ruleref uri="#items"/>
    <tag>assign(i $return)</tag>
  </item>
  <tag><![CDATA[<number $n> <item $i>]]></tag>
</rule>

<rule id="items" scope="public">
  <one-of>
    <item>
      <one-of>
        <item>helmet</item>
   <item>helmets</item>
      </one-of>
    </item>
    <item>
```

```
      <one-of>
        <item>stick</item>
    <item>sticks</item>
      </one-of>
     </item>
     <item>
      <one-of>
        <item>puck</item>
    <item>pucks</item>
      </one-of>
     </item>
     <item>
      <item repeat="0-1">
        <one-of>
       <item>pair of</item>
       <item>pairs of</item>
    </one-of>
      </item>
      <item repeat="0-1">hockey</item>
        <one-of>
       <item>pant</item>
       <item>pants</item>
    </one-of>
     </item>
     <item>
      <item repeat="0-1">
        <one-of>
       <item>pair of</item>
       <item>pairs of</item>
    </one-of>
      </item>
      socks
     </item>
     <item>
      <item repeat="0-1">
        <one-of>
       <item>pair of</item>
       <item>pairs of</item>
    </one-of>
      </item>
      skates
     </item>
     <item>
      <one-of>
        <item>jersey</item>
    <item>jerseys</item>
      </one-of>
     </item>
  </one-of>
  <tag>return($string)</tag>
</rule>

<!-- Simple number grammar -->
```

```
<rule id="number">
  <one-of>
    <item>one   <tag>return(1)</tag></item>
    <item>two   <tag>return(2)</tag></item>
    <item>three <tag>return(3)</tag></item>
    <item>four  <tag>return(4)</tag></item>
    <item>five  <tag>return(5)</tag></item>
    <item>six   <tag>return(6)</tag></item>
    <item>seven <tag>return(7)</tag></item>
    <item>eight <tag>return(8)</tag></item>
    <item>nine  <tag>return(9)</tag></item>
  </one-of>
</rule>

<!-- Fillers -->
<rule id="pre_filler">
  <one-of>
    <item>i would like</item>
    <item>i'd like</item>
    <item>i want</item>
    <item>one</item>
  </one-of>
  to
  <one-of>
    <item>buy</item>
    <item>get</item>
    <item>purchase</item>
  </one-of>
</rule>

<rule id="post_filler">
  <one-of>
    <item>please</item>
  </one-of>
</rule>

</grammar>
```

Sample YesNo grammar

The following code shows two sample grammars, *YesNo* and *StrictYesNo*.

- *YesNo* handles loose expressions, for example, "Yes," "that's correct," "thank you."
- *StrictYesNo* is limited to one-word responses.

```
<?xml version="1.0" ?>
<grammar xmlns="http://www.w3.org/2001/06/grammar"
         xmlns:nuance="http://voicexml.nuance.com/grammar"
         version="1.0" xml:lang="en" root="YesNo">

<!-- Public grammars -->

<!-- Loose YesNo grammar -->
```

```
<rule id="YesNo" scope="public">
  <item repeat="0-1" repeat-prob="0.01">
    <ruleref uri="#YesNo_PreHesitation"/>
  </item>
  <ruleref uri="#YesNo_CORE"/>
</rule>

<!-- Strict YesNo grammar -->
<rule id="StrictYesNo" scope="public">
  <ruleref uri="#StrictYesNo_CORE"/>
</rule>

<!-- Helper subgrammars -->

<!-- Prehesitation -->
<rule id="YesNo_PreHesitation">
  <one-of>
    <item>um</item>
    <item>uh</item>
    <item>hm</item>
  </one-of>
</rule>

<!-- Core grammars -->
<rule id="YesNo_CORE">
  <one-of>
    <item>
      <ruleref uri="#YesLoose"/>
      <tag><![CDATA[<YesNo yes>]]></tag>
    </item>
    <item>
      <ruleref uri="#NoLoose"/>
      <tag><![CDATA[<YesNo no>]]></tag>
    </item>
  </one-of>
</rule>

<rule id="StrictYesNo_CORE">
  <one-of>
    <item>
      <ruleref uri="#YesStrict"/>
      <tag><![CDATA[<YesNo yes>]]></tag>
    </item>
    <item>
      <ruleref uri="#NoStrict"/>
      <tag><![CDATA[<YesNo no>]]></tag>
    </item>
  </one-of>
</rule>

<!-- Subgrammar used in YesNo -->
<rule id="YesLoose">
  <one-of>
```

```
<item>yes please</item>
<item>
  yes
  <item repeat="0-1">
    it
    <item repeat="0-1">
      <one-of>
        <item>sure</item>
        <item>certainly</item>
      </one-of>
    </item>
    is
  </item>
</item>
<item>
  it
  <item repeat="0-1">
    <one-of>
      <item>sure</item>
      <item>certainly</item>
    </one-of>
  </item>
  is
</item>
<item>yup</item>
<item>yeah</item>
<item>okay</item>
<item>sure</item>
<item>you got it</item>
<item>
  <item repeat="0-1">
    <item repeat="0-1">yes</item>
    <one-of>
      <item>that's</item>
      <item>it's</item>
      <item>that is</item>
      <item>it is</item>
    </one-of>
  </item>
  <one-of>
    <item>right</item>
    <item>correct</item>
  </one-of>
</item>
<item>
  <item repeat="0-1">
    <one-of>
      <item>yes</item>
      <item>yeah</item>
    </one-of>
  </item>
  i
  <one-of>
```

```
        <item>would</item>
        <item>do</item>
      </one-of>
    </item>
  </one-of>
  <item repeat="0-1">
    <one-of>
      <item>thanks</item>
      <item>thank you</item>
    </one-of>
  </item>
</rule>

<!-- Subgrammar used in StrictYesNo -->
<rule id="YesStrict">
  <one-of>
    <item>yes</item>
    <item>yup</item>
    <item>yeah</item>
    <item>right</item>
    <item>correct</item>
  </one-of>
</rule>

<!-- Subgrammar used in YesNo -->
<rule id="NoLoose">
  <one-of>
    <item>nope</item>
    <item>absolutely not</item>
    <item>
      no
      <item repeat="0-1">way</item>
    </item>
    <item>
      <item repeat="0-1">no</item>
      <one-of>
        <item>it isn't</item>
        <item>it's not</item>
        <item>it is not</item>
      </one-of>
    </item>
    <item>
      <item repeat="0-1">
        <item repeat="0-1">no</item>
        <one-of>
          <item>that's</item>
          <item>it's</item>
          <item>that is</item>
          <item>it is</item>
        </one-of>
      </item>
      <one-of>
        <item>wrong</item>
        <item>
          not
          <one-of>
            <item>correct</item>
            <item>right</item>
          </one-of>
        </item>
        <item>incorrect</item>
      </one-of>
    </item>
    <item>
      <item repeat="0-1">no</item>
      i
      <one-of>
        <item>would not</item>
        <item>wouldn't</item>
        <item>do not</item>
        <item>don't</item>
      </one-of>
    </item>
  </one-of>
  <item repeat="0-1">
    <one-of>
      <item>thanks</item>
      <item>thank you</item>
    </one-of>
  </item>
</rule>

<!-- Subgrammar used in StrictYesNo -->
<rule id="NoStrict">
  <one-of>
    <item>no</item>
    <item>nope</item>
    <item>wrong</item>
    <item>incorrect</item>
  </one-of>
</rule>

</grammar>
```

Analysis and Tuning Tools: Reference information

D

This appendix provides the DTDs for the grammar overrides file and the scores file.

DTD for the grammar overrides file

This section provides the DTD for the grammar overrides file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!--
Nuance Analysis and Tuning Tools
Grammar Overrides DTD (20030922)
-->

<!-- root element -->
<!ELEMENT grammar-overrides (default-parameters?, group*)>

<!--
default-parameters element. Specifies a set of default values of
parameters for an experiment. May be overridden by parameters for a group
defined within this overrides file.
-->

<!ELEMENT default-parameters (param*)>

<!--
group element. Specifies the grammars and parameters that should be used
for all instances of grammar-defs within a group.
-->

<!ELEMENT group grammar-def>
<!ATTLIST group
        name CDATA #REQUIRED>
```

```
<!--
grammar-def element. Specifies the grammar, dynamic grammar insertions
(optional), and parameters for a specific instance of a grammar definition
within a group.
-->

<!ELEMENT grammar-def (grammar, dynagrams?, parameters)>
<!ATTLIST grammar-def
        instance CDATA #REQUIRED>

<!--
grammar element. Specifies the grammar that should be used for recognition
for a specific grammar definition. May be a static grammar (.MAIN) or
just-in-time grammar source.
-->

<!ELEMENT grammar (#PCDATA)>

<!--
dynagrams element. Specifies the dynamic grammars, by key, in a dynamic
grammar database, that should be inserted into the recognition package at
a specific label.
-->

<!ELEMENT dynagrams (insert*)>

<!ELEMENT insert EMPTY>
<!ATTLIST insert
        key CDATA #REQUIRED
        label CDATA #REQUIRED>

<!--
parameters element. Specifies the set of recognition parameters that
should be set for a specific grammar definition.
-->

<!ELEMENT parameters (param*)>

<!--
param element. Specifies a single recognition parameter. The name must be
a valid Nuance parameter name, e.g., rec.Pruning
-->

<!ELEMENT param EMPTY>
<!ATTLIST param
        name CDATA #REQUIRED
        value CDATA #REQUIRED>
```

DTD for the scores file

This section provides the DTD for the scores file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!--
Nuance Analysis and Tuning Tools
Scores Report DTD (20030922)
-->

<!-- root element -->
<!ELEMENT report (call*)>
<!ATTLIST report
        name CDATA #FIXED "scores">

<!--
call element. Provides details on a specific call that was analyzed and
scored.
-->

<!ELEMENT call (datum*)>
<!ATTLIST call
        log CDATA #REQUIRED>

<!--
datum element. Provides details on a specific piece of data within a call.
-->

<!ELEMENT datum (record, transcription, nl-ref?, score)>
<!ATTLIST datum
        group CDATA #REQUIRED>

<!--
record element. Provides information about the call log record that was
analyzed and scored to produce a datum.
-->

<!ELEMENT record (fields, recresult)>
<!ATTLIST record
        gdef-instance CDATA #REQUIRED
        utt CDATA #REQUIRED>

<!-- fields element. Contains a set of fields from a call log record. -->

<!ELEMENT fields (field+)>

<!--
field element. Contains the value for a single field from a call log
record.
-->

<!ELEMENT field (#PCDATA)>
<!ATTLIST field
        name CDATA #REQUIRED>

<!--
```

```
recresult element. Contains the recresult for a datum. This recresult is
either based on the call logs for the deployed application, or on the
results of the recognition experiment being run.
-->

<!ELEMENT recresult (result*)>
<!ATTLIST recresult
        type CDATA #REQUIRED
        top-conf CDATA #IMPLIED>

<!--
result element. Single hypothesis in the N-best list of a recresult. The
value is the nl for the result; the recog attribute contains the
recognized text.
-->

<!ELEMENT result (#PCDATA)>
<!ATTLIST result
        recog CDATA #REQUIRED>

<!--
transcription element. The transcription for a datum. The original
transcription and labels from the input transcriptions file are contained
in the attributes, as is the cleaned up version of the transcription.
-->

<!ELEMENT transcription EMPTY>
<!ATTLIST transcription
        orig CDATA #REQUIRED
        clean CDATA #REQUIRED
        labels CDATA #REQUIRED>

<!--
nl-ref element. The nl interpretation(s) of the clean transcription.
-->

<!ELEMENT nl-ref (#PCDATA)>

<!--
score element. The scoring information for a datum. This information is
contained in the following attributes:
- acc-stat: statistic labeling this datum as CA-in, FA-in, FR-in, CR-out,
FA-out. based on only the first hyp, using the intended
rec.ConfidenceRejectionThreshold.
- acc-stat-no-rej: accuracy statistic based on no rejection, rather than
using the intended value of rec.ConfidenceRejectionThreshold. Only
meaningful in experiments.
- ig-stat: Statistic labeling this datum as in grammar (IG), out of
grammar (OOG), or out of coverage (OOC). out of coverage is a subset of
OOG.
- top-correct: index (0-based) of which n-best result is CA-in. If none of
the results were CA-in, the value is none. -->
```

```
<!ELEMENT score EMPTY>
<!ATTLIST score
        acc-stat (CA-in | FA-in | FR-in | CR-out | FA-out) #REQUIRED
       acc-stat-no-rej (CA-in | FA-in | FR-in | CR-out | FA-out) #REQUIRED
        ig-stat (IG | OOG | OOC) #REQUIRED
        top-correct CDATA #REQUIRED>
```

DTD for the records file

This section provides the DTD for the records file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!--
Nuance Analysis and Tuning Tools
Records Report DTD (20030922)
-->

<!-- root element -->
<!ELEMENT report (call*)>
<!ATTLIST report
        name CDATA #FIXED "records">

<!--
call element. Provides details on a specific call that was analyzed.
-->

<!ELEMENT call (record*)>
<!ATTLIST call
        log CDATA #REQUIRED>

<!--
record element. Provides information about the call log record that was
analyzed.
-->

<!ELEMENT record (fields, recresult)>
<!ATTLIST record
        group CDATA #REQUIRED
        gdef-instance CDATA #REQUIRED
        utt CDATA #REQUIRED>

<!-- fields element. Contains a set of fields from a call log record. -->

<!ELEMENT fields (field+)>

<!--
field element. Contains the value for a single field from a call log
record.
-->
```

```
<!ELEMENT field (#PCDATA)>
<!ATTLIST field
        name CDATA #REQUIRED>

<!--
recresult element. Contains the recresult for a datum. This recresult is
either based on the call logs for the deployed application, or on the
results of the recognition experiment being run.
-->

<!ELEMENT recresult (result*)>
<!ATTLIST recresult
        type CDATA #REQUIRED
        top-conf CDATA #IMPLIED>

<!--
result element. Single hypothesis in the N-best list of a recresult. The
value is the nl for the result; the recog attribute contains the
recognized text.
-->
<!ELEMENT result (#PCDATA)>
<!ATTLIST result
        recog CDATA #REQUIRED>
```

DTD for the dataset file

This section provides the DTD for the records file:

```
<!-- root element -->
<!ELEMENT dataset (call*)>

<!-- call element. Specifies a call log that should be processed. -->
<!ELEMENT call (utt*)>
<!ATTLIST call
   log CDATA #REQUIRED
   utts (all|selected) "all">

<!-- utt element. Specifies an utterance within a call that should be -->
<!-- processed. -->
<!ELEMENT utt EMPTY>
<!ATTLIST utt
   filename CDATA #REQUIRED>
```

Appendix E

Nuance transcriptions conventions

E

To run recognition experiments, you need to provide *transcriptions*—written text corresponding to the utterances spoken by the caller. This appendix describes how to provide transcriptions that follow the standard Nuance transcriptions conventions.

Note: Nuance offers a transcription service to help you generate transcriptions for your utterances. Please contact Nuance Professional Services for more information.

Overview

Transcriptions are provided in one or more transcription files. At a minimum, a transcription file contains the following information for each utterance:

```
FILE: path to utterance file
TRANSCRIPTION: transcription for the utterance
```

For example, consider the utterance "tomatoes and mushrooms" for the PizzaTalk application. The transcriptions file might look as follows for this utterance:

```
FILE: 2003/02February/20/12/09-29-zamboni-001_5060-utt02.wav
TRANSCRIPTION: tomatoes and mushrooms
```

Note that the path in the `FILE` field can be absolute or relative.

The `TRANSCRIPTION` field can also contain additional information to indicate noises or speech anomalies. This information is provided using markers, such as the `[side_speech]` marker. For example, if the caller coughs before saying "tomatoes and mushrooms" for the PizzaTalk application, the transcriptions file might look as follows for this utterance:

```
FILE: 2003/02February/20/12/09-29-zamboni-001_5060-utt02.wav
```

```
TRANSCRIPTION: [side_speech] tomatoes and mushrooms
```

You can also specify a `LABELS` field to indicate an attribute of the utterance, for example, to indicate whether the speaker was male or female. If the caller in the example above is a woman, the transcriptions file might look as follows for this utterance:

```
FILE: 2003/02February/20/12/09-29-zamboni-001_5060-utt02.wav
TRANSCRIPTION: [side_speech] tomatoes and mushrooms
LABEL: female
```

You then repeat the `FILE`, `TRANSCRIPTION`, and `LABELS` (if available) fields for each utterance; for example:

```
FILE: 2003/02February/20/12/09-29-zamboni-001_5060-utt01.wav
TRANSCRIPTION: how about a large pizza
FILE: 2003/02February/20/12/09-29-zamboni-001_5060-utt02.wav
TRANSCRIPTION: tomatoes and mushrooms
FILE: 2003/02February/20/12/09-29-zamboni-001_5060-utt03.wav
TRANSCRIPTION: yeah
```

Providing the transcription in the TRANSCRIPTION field

You provide the transcription in the `TRANSCRIPTION` field. This section first describes the writing conventions to use for the transcription. It also describes markers that you can add to the `TRANSCRIPTION` field to indicate noise events and speech anomalies.

Text conventions

Follow these text conventions for the `TRANSCRIPTION` field:

- All text should be written in lowercase
- **Initials**: Write as letters separated by spaces
- **Numbers**: Write out in words. For example: `five five five one two three four`
- **Punctuation marks**: You can use apostrophes (o'clock, todd's), but don't use any other punctuation (for example, !.,:;?"-)
- **Abbreviations**: Don't use them. For example, use `mister`, not `mr`
- **Money amounts**: Write out as words. For example: `five dollars and fifty cents`

- **Spelled words**: Write out as lower case letters separated by spaces. For example: `c e n t r a l`
- **Proper names**: Spell them as best as you can

Noise events

Used in the `TRANSCRIPTION` field, noise events don't overlap with speech in time. They are denoted by brackets (`[]`). If noises occur during speech, then use the `speech-in-noise` label (see "`speech-in-noise`" on page 238) and not the noise event (`[]`) marker.

The following table describes the valid noise event markers:

Marker	Description	Example
`[side_speech]`	Speech that is not intended for the recognizer, for example, peripheral speech, prompt speech, and so on. Includes laughter and other vocally generated mouth noises such as sneezing, coughing, and clearing of the throat. This also includes speech that is actually the system prompt, which hasn't been properly removed by the echo canceller (for example, "please say the city you are flying to").	**Utterance**: Coughs then says "Customer service" **Transcription**: `[side_speech] customer service`
`[breath_noise]`	Inhale and exhale only. Not clicks or other mouth noises.	**Utterance**: Says "customer service" then exhales **Transcription**: `telephone service [breath_noise]`
`[hang_up]`	Click from phone being hung up.	**Utterance**: Says "phone bill", coughs, then hangs up **Transcription**: `phone bill [side_speech] [hang_up]`

Marker	Description	Example
`[dtmf]`	DTMF tone.	**Utterance**: Says “phone bill” then presses DTMF button **Transcription**: `phone bill [dtmf]`

`[fragment]`	A word that is fragmented or interrupted. The waveform may or may not be truncated. A word may be fragmented because the speaker stopped speaking, the speaker paused partway through a word, the speaker stuttered, or the system truncated the start or end of an utterance.	**Utterance**:
		Says “good mor- morning”
		Transcription:
		`good [fragment] morning`
		Utterance:
		Says “good mor- <pause> -ning”
		Transcription:
		`good [fragment] [fragment]`

`[noise]`	All other noise events that don't overlap with speech.	**Utterance**: Car beeps than caller says “billing” **Transcription**: [noise] billing

Speech anomalies

Speech anomalies are also used in the `TRANSCRIPTION` field. The following table describes the valid speech anomalies markers:

Marker	Description
`(())`	Unintelligible word(s). Any word(s) that the transcriber cannot understand. For example: `TRANSCRIPTION : [noise] (())`
`((word))`	Unintelligible words with the transcriber's best guess. Any utterance that the transcriber cannot clearly understand but for which a best guess can be provided. If more than one word are unintelligible, put each word in between double parentheses`(())`. For example: `TRANSCRIPTION : [noise] i’m all ((done))`

Marker	Description
`*word*`	Mispronounced word. Allowances are made for accents. Correct spelling of the word is placed in between stars `* *`. For example: `TRANSCRIPTION : [noise] *help*`
`~`	Truncation of waveform, leading to cutting off of speaker's speech. If the waveform is truncated at the beginning, ~ is placed at the beginning of the transcription; if truncation occurs at the end, ~ is placed at the end. The truncation marker is always used in conjunction with the `[fragment]` noise event marker. For example: `TRANSCRIPTION : ~ [fragment] balance`
`@hes@`	Hesitation word; for example, “um”, “uh”, “ah”, “eh”, “ehn”, “hm”. Use `@hes@` only for fillers with no meaningful counterparts and approximate a central vowel (vocalization of the neutral open mouth position) or a syllabic nasal (vocalization of the closed mouth position), such as ones exemplified above. Do not use `@hes@` for filler words that correspond to meaningful words, for example, “well” in English, or “ano” in Japanese. For example: `TRANSCRIPTION : @hes@ visual i guess`

Specifying labels with the LABELS field

Labels indicate an attribute of the utterance that is (relatively) independent of time. You specify labels in the `LABELS` field. If you specify more than one label, simply separate them with a space. For example:

```
LABELS : female speech-in-noise non-native
```

The following table describes the valid labels:

Label	Description
`male` `female`	Gender of caller.
`non-native`	A strong, non-native accent in the sense that the speaker doesn't speak the language natively. For example, a French person speaking English could be non-native. But a Bolivian speaking Spanish to an application deployed in Argentina is not non-native because the Bolivian speaks Spanish natively.

Label	Description
`bad-audio`	Corrupted signal, such as very loud static, speech heavily distorted, or speech containing an echo of the speaker. This can also include audio clipped in mid utterance (NOT at the beginning or end), like when a digital cell phone momentarily drops the signal, resulting in "missing" sound.
`speech-in-noise`	Background noise occurring while speaker is speaking. May include music playing, phone ringing, baby crying, or people talking in the background.
`no-waveform`	The waveform file doesn't exist.
`no-speech`	The waveform file exists but there's no speech in it.

Index