

NUANCE

**Nuance Speech Recognition System
Version 7.0**

Grammar Developer's Guide

Nuance Speech Recognition System, Version 7.0
Nuance Grammar Developer's Guide

Copyright © 1996-2001 Nuance Communications, Inc. All rights reserved.
1005 Hamilton Avenue, Menlo Park, California 94025 U.S.A.
Printed in the United States of America.

Information in this document is subject to change without notice and does not represent a commitment on the part of Nuance Communications, Inc. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. You may not copy, use, modify, or distribute the software except as specifically allowed in the license or nondisclosure agreement. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose, without the express written permission of Nuance Communications, Inc.

Nuance and Nuance Communications are registered trademarks of Nuance Communications, Inc. SpeechObjects, SpeechChannel, and Verifier are trademarks of Nuance Communications, Inc. Any other trademarks belong to their respective owners.

Contents

About this guide	vii
Using this manual	vii
Nuance developer documentation	viii
Developer guides	viii
Online documentation	ix
Typographical conventions	ix
Where to get help	x
Chapter 1. The grammar development process	1
Principles of effective grammar design	1
Programming versus grammar writing	1
Predicting how callers will speak	2
Steps in the development process	3
Define the dialog	3
Identify the information items and define the slots	4
Design the prompts	5
Anticipate the caller responses	6
Identify the “core” of the grammar	7
Identify the “filler” portion of the grammar	8
Define your grammar in the Grammar Specification Language	9
Add natural language commands	9
Building a recognition package	10
Chapter 2. Defining grammars	13
Creating a grammar file	13
A first example using Nuance Grammar Builder	13
The Grammar Specification Language	16
Grammar names	17

Grammar descriptions	17
Writing grammars with Nuance Grammar Builder	19
Grammar hierarchies	19
Top-level grammars and subgrammars	20
Including grammar files	21
The NGB grammar library	23
Recursive grammars	24
Grammar probabilities	25
Assigning probabilities within NGB	25
Probabilities in GSL	26
Dynamic grammars	27
Dynamic grammar specification	28
Dynamic grammar referencing	29
GSL syntax for enrollment	29
Language-specific grammar conventions	30
Chapter 3. Compiling grammars	31
Compiling a grammar with NGB	31
<i>nuance-compile</i> inputs	31
<i>nuance-compile</i> output	32
Choosing a model set	33
Acoustic model naming convention	33
Language-specific models	34
Support for multiple languages	35
Adding missing pronunciations	35
Creating custom dictionaries within NGB	36
Creating a dictionary file	36
Merging and overriding dictionaries	37
Automatic pronunciation generator	37
Phrase pronunciations	38
Optimizing recognition packages	39
Creating unflattened grammars	40
Speeding up compilation	40
Filler grammars	40

Tuning recognition performance	41
Summary of compiler options	42
Chapter 4. Natural language understanding	45
Defining natural language interpretations	46
Natural language commands	46
The slot definitions file	50
Compiling a grammar with natural language support	51
Ambiguous grammars	51
Advanced features	52
Functions	52
Complex values	55
Chapter 5. Testing grammars	61
Grammar testing overview	61
Testing a grammar with Nuance Grammar Builder	62
Creating an initial test set	62
Step-by-step grammar testing instructions	63
Regression tests	68
Command-line tools for grammar testing	68
<i>parse-tool</i>	69
<i>generate</i>	69
<i>nl-tool</i>	70
<i>Xapp</i>	70
<i>batchrec</i>	71
Chapter 6. Testing recognition performance	73
Choosing <i>batchrec</i> test sets	74
Using <i>batchrec</i>	74
Creating the testset file	75
Optional arguments	78
Setting Nuance parameters	81
Output from <i>batchrec</i>	82

Chapter 7. Creating application-specific dictionaries	85
The Nuance phoneme set	85
Multiple pronunciations	91
Sample dictionary file	92
Converting non-CPA dictionaries	94
Phoneme sets for other languages	94
Appendix A. GSL reference	95
GSL syntax	95
Summary of package files	97

About this guide

This manual provides guidelines for developing and testing a grammar, and the tools that Nuance offers for building efficient recognition packages. The guide's focus is on the graphical tool Nuance Grammar Builder (NGB) and how you can use it to build recognition packages.

Nuance Grammar Builder provides a single environment with a rich graphical user interface that lets you create, compile, and test grammar packages. This graphical tool is equipped with a complete help system that shows how to use its features effectively. This tool runs on the Windows operating system only.

In addition, this guide discusses the command line utilities available for grammar development. You can use these instead of NGB if you are working in a UNIX environment or if you prefer a command-line interface.

Using this manual

This manual presents detailed 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

This manual 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 describes how to compile a grammar into a recognition package and append additional word pronunciations, and discusses some optimization and performance issues.

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 presents a comprehensive way of testing all aspects of a grammar, including coverage, interpretation, ambiguity, and regression tests.

Chapter 6 describes how to use the Nuance *batchrec* utility to test the performance of recognition packages.

Chapter 7 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.

Nuance developer documentation

The Nuance documentation includes a set of developer guides as well as comprehensive online API reference documentation.

Developer guides

In addition to the *Nuance Grammar Developer's Guide*, the documentation set includes:

- *Introduction to the Nuance System*, which provides a comprehensive overview of the Nuance System, the available tools and interfaces, and the development process. This guide is for both speech application developers and integrators. It provides background information relevant for both types of developers and discusses the design and development processes.
- The *Nuance Application Developer's Guide* describes how to develop, configure, and tune a Nuance speech application. This manual describes features provided by the Nuance System, such as dynamic grammars and hot word recognition, and describes general application development and deployment issues such as tuning Nuance parameter settings, using Nuance audio providers, and launching recognition clients, servers, and other required processes. This manual also describes the Dialog Builder, a Nuance C API you can use for prototyping speech applications. Documentation for other application development APIs, such as the Foundation

SpeechObjects™, is packaged with those APIs, which are currently released separately.

- The *Nuance Platform Integrator's Guide*, which describes the process, requirements, and Nuance APIs for integrating the Nuance System with an existing IVR platform or toolkit. The Nuance integration APIs include the RCEngine, a C++ class that provides access to Nuance recognition, recording, playback, and telephony control features, and the VRecServer, a lower-level C++ class that provides access to Nuance recognition but lets you use your own audio mechanisms. The Nuance System also includes the RCAPI, a set of C functions that provide functionality analogous to the RCEngine.
- The *Nuance Verifier Developer's Guide* describes how to use the Nuance Verifier™ to provide secure access to transactions and sensitive information through voice identification. The Nuance Verifier, which is licensed separately from the Nuance Speech Recognition System, allows applications to verify the identity of a caller by analyzing the caller's voiceprint.
- Foundation SpeechObjects documentation, including the *SpeechObjects Developer's Guide*, which describes the Nuance SpeechObjects framework and how to use it to build a speech recognition application. This guide also provides detailed information on using each of the Foundation SpeechObjects and on using the speaker verification classes. This documentation is shipped with the Foundation SpeechObjects product.

Online documentation

The product installation includes HTML-based documentation for Nuance APIs, parameters, command-line utilities, and GSL (grammar specification language) syntax. To access this documentation open the file `%NUANCE%\doc\api\index.html`.

The `%NUANCE%\doc` folder also contains an online copy of all Nuance developer manuals in PDF format. To view the entire set of online documentation, open the file `%NUANCE%\index.html` in any HTML browser. Install this file and the `%NUANCE%\doc` directory on an internal web server or file system to minimize space requirements.

Typographical conventions

The Nuance manuals use the following text conventions:

<i>italic</i>	Used to indicate variables, file and path names, program names, and terms introduced for the first time.
Courier	Used to indicate class, function, and parameter names.
> Courier	Indicates commands or character 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.

Note: The Nuance System and SpeechObjects run 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. Where usage or functionality differs on Windows and UNIX platforms, differences are specifically called out.

Where to get help

If you have questions or problems, Nuance provides technical support through the Nuance Developer Network™ (NDN), a web-based resource center that includes online forums, technical support guides on specific topics, access to software updates, as well as a support request form. If you are a member, go to *extranet.nuance.com* to log on, or see the Nuance website (*www.nuance.com*) for information on how to become a member.

If you have comments on the documentation please send e-mail directly to *techdoc@nuance.com*.

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, because of well-designed prompts and grammars, the user's utterances fall largely within an expected set of phrases, allowing 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.

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. Chapter 2 describes the syntactic structure and elements of grammars in detail.

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. As a rule of thumb, a 5% out-of-grammar (OOG) rate is considered acceptable. Even 10-20% OOG rates are not uncommon for certain types of grammars.

Note: A phrase is *out-of-grammar* if it cannot be parsed by a grammar.

A good piece of 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 and recommendations on ways to format the encoding of a grammar are included in Appendix A.

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 Anticipate the caller responses
- 5 Identify the “core” and “filler” portions of your grammars
- 6 Write the GSL code for your grammars
- 7 Add natural language commands
- 8 Build a recognition package

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 detail 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-mode dialog?

The answers to these questions determine the shape and content of the grammars you need to develop.

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 certain type. (See the introduction to Chapter 4 for more 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). You might then ask the caller if they want to hear the return flight information, start over, or hang-up (a small set of commands). That's six 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 table like the following:

Item	Slot name	Value format	Value type
city #1	origin	3-letter code	string
city #2	destination	3-letter code	string
date	date	[<month> <day>]	NL structure
time	time	0-2359	integer
yes/no	confirm	"yes" or "no"	string
restart/hangup	command	"restart" or "hangup"	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 and information items, 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 being 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?	destination
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
Would you like to start over or hang up?	command

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, although it is doable. It is up to you to decide whether the dialog will elicit only simple responses or more complex ones.

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.

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.

<i>What city would you like to leave from?</i>	
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	
(I'm) departing from San Francisco	[some additional possibilities]

<i>What city would you like to fly to?</i>	
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”]
(I'm) going to New York	[some additional possibilities]
My destination is New York	

<i>What date would you like to leave?</i>	
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]
Leaving May second	[a third literal response]
Um, May second, please	[hesitation + final “please”]

<i>What time would you like to depart?</i>	
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"]

<i>You're going from <origin> to <dest> on <date> at <time>. Is this correct?</i>	
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 (or yup, or you bet)	[casual alternatives]

<i>Would you like to start over or hang up?</i>	
Start over	[command by itself]
Hang up	[command by itself]
I'd like to start over	[a literal response]
Um, start over please	[hesitation + final "please"]

Identify the “core” of the grammar

A grammar typically consists of a “core” portion that contains the most important meaning-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—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, time, and confirmation. The “start over” and “hang up” commands are instead specific to this application, so no core grammar need be created for these.

The Grammar Library folder in Nuance Grammar Builder (NGB) contains sample grammars for a variety of “core” grammars. You can use them as-is or modify them to suit your particular needs. Most likely, your application will need a “core” grammar that is not provided with the Grammar Library and that you will have to write entirely. However, the grammars shipped with NGB can still serve as models for structure and coding standards.

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 tokens *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

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.

Define your grammar in the Grammar Specification Language

The Grammar Specification Language (GSL) is the language you use to formally specify a grammar for a Nuance System application.

The two grammars in the flight information example (departure city and date), are readily translated to GSL from the lists above. Assuming that you have the *CITY* and *DATE* subgrammars (for example, from the Grammar Library), the code looks 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)
]
```

CITY and *DATE* are subgrammars defined elsewhere.

Add natural language commands

The next step, adding natural language commands to the grammar, is straightforward, but you need to know how to do it using GSL. 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>}
```

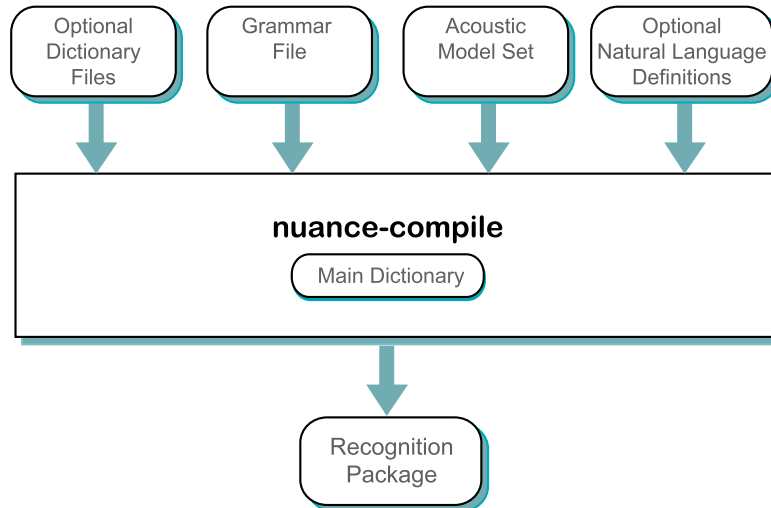
The grammars are now ready to be compiled and tested.

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 acoustic models 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 within NGB or with the command-line program *nuance-compile*.



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

- A grammar file name

- An acoustic model 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 acoustic model set defines, among other things, the languages that an application will use. Models differ in their accuracy, memory requirements, and computational requirements. Acoustic models are discussed in Chapter 3.

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 3.

Defining 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 weighs 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 explains how to write simple grammars. Chapter 4 describes more complex grammar-writing techniques that include natural language interpretation.

Creating a grammar file

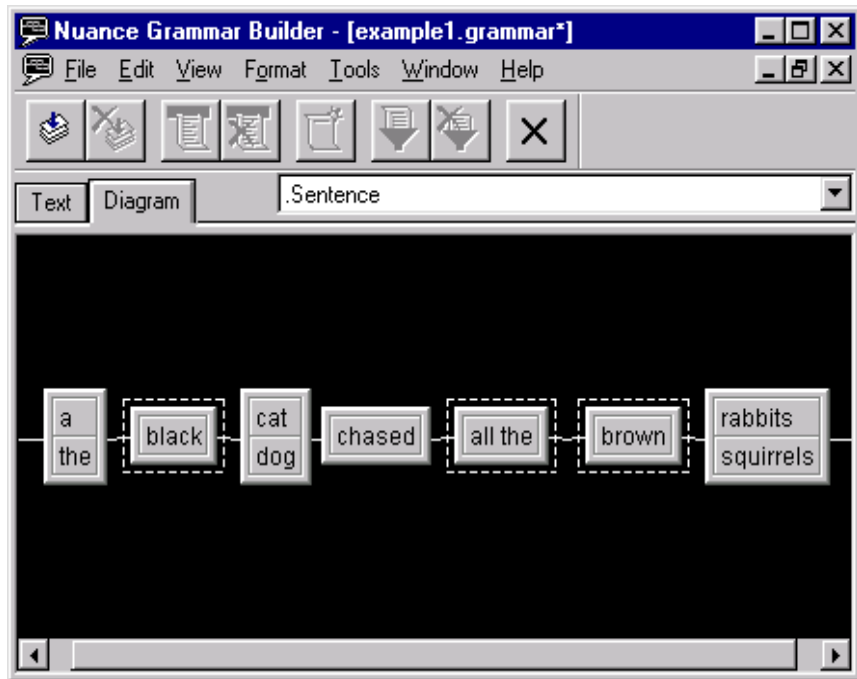
You specify a grammar in a *grammar file*. A grammar file is a text file—it can contain more than one grammar definition, and it has the file extension *.grammar*; for example, *myApp.grammar*. A grammar file is the basic component necessary to build a recognition package.

A first example using Nuance Grammar Builder

Before plunging into the details of grammar writing, this section presents a very simple grammar and the different views of that grammar within the NGB’s environment. This grammar describes sentences like “a cat chased rabbits,” “a black dog chased all the brown squirrels,” and “the black cat chased squirrels,” using the Grammar Specification Language (GSL). GSL syntax is described in more detail later in this chapter.

The screen shot below shows the main NGB frame and a file—named *example.grammar*—that specifies a very simple grammar. In this shot, the

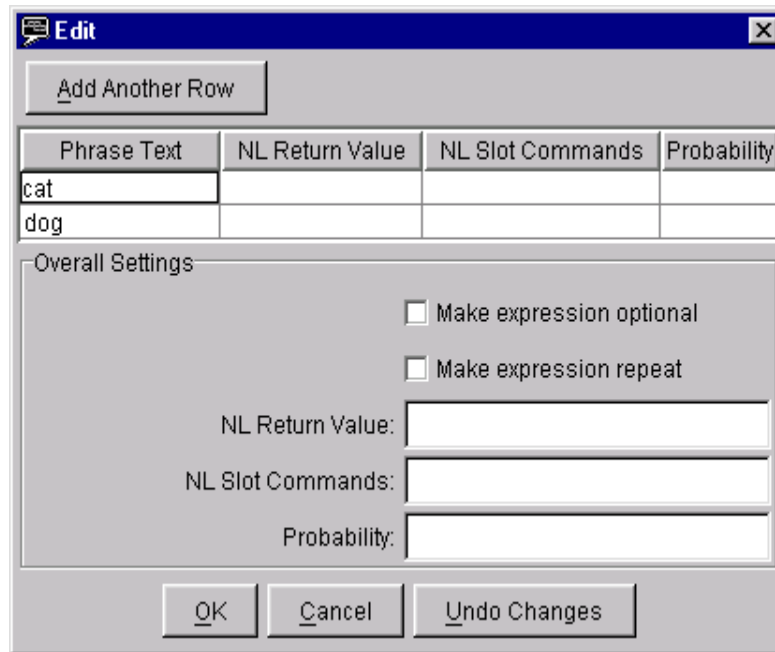
Diagram tab of the window shows the graphical representation of a grammar named *.Sentence*.



Each box in the diagram represents a grammar construct:

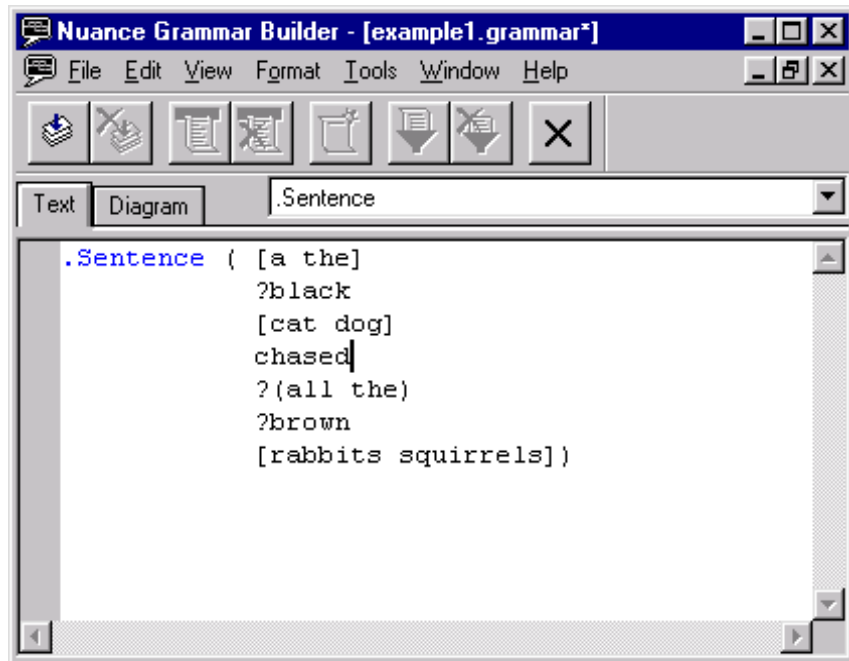
- Optional constructs—like *black*—are enclosed in a dotted white box
- Sequential constructs—like the components of one whole sentence—are linked from left to right
- Alternate constructs—like *cat/dog*—are represented vertically stacked

Double-clicking a box—which represents a grammar expression—brings up a dialog like the following that lets you set properties for that expression.



The dialog shown corresponds to the alternate expression “*cat/dog*”. You can set a variety of attributes of this construct including making it optional, repeating it one or more times, assigning a probability to it, adding alternates, and attaching natural language interpretations to each of its alternates.

Clicking the Text tab of the window displays the specification of the *same* grammar in GSL, as shown in the next screen shot.



The name of the grammar described in both views is *.Sentence*. The pull down list at the top right of the window—displaying *.Sentence* in both window screen shots—contains the list of all the grammars in the opened file. Each grammar can be separately displayed and edited.

The Grammar Specification Language

You write grammar definitions using a special language called the *Grammar Specification Language*, or *GSL*.

At the top level a grammar definition has the format:

GrammarName GrammarDescription

where *GrammarName* 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

GrammarName 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 17). 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 2. 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 2: Grammar operators

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

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

Writing grammars with Nuance Grammar Builder

While you can define grammars by simply creating and editing text files, NGB assists you in writing your grammars in several ways:

- You can display a grammar specification in either a text or a diagram view:
 - The text view shows the GSL definition of the grammar
 - The diagram view shows a graphical representation on the grammar, often making it easier to understand the structure of the grammar

You can switch back and forth between these views.

- The text view editor supports “syntax coloring,” a feature that highlights various GSL constructs.
- In the diagram view, the GSL operators for concatenation, disjunction, optionality, and positive and kleene closure are visually represented. You can expand or collapse an expression to have a detailed or global view of the grammar using that expression.
- In either view, you can display just one grammar from all those described in a grammar file.
- Both views have editing capabilities. The changes made in one view are instantly reflected in the other.

See the NGB help system for a complete description of all the features that support grammar writing.

Grammar hierarchies

You can build a hierarchy of grammars using *subgrammars*. 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

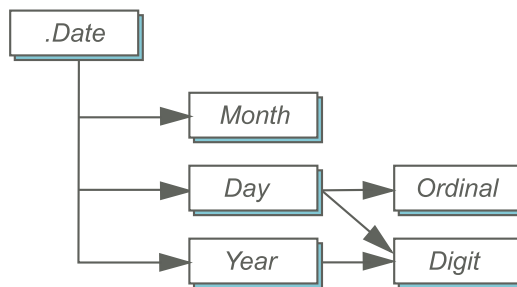
- Helps focus the grammar development to the task at hand
- Hides unnecessary details and promotes modularity

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. You could think of a top-level grammar as being “public,” while a subgrammar is “private”—only accessible within the package.

Caution: The distinction between top grammars and subgrammars does not apply to grammars used by Nuance SpeechObjects. See the SpeechObjects documentation for details on SpeechObjects-specific grammar conventions.

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 use during 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 saying one of the non-zero digits
NZDIGIT [one two three four five six seven eight nine]
; subgrammar for saying one of the digits including zero
DIGIT [one two three four five six seven eight nine zero oh]
; subgrammar for saying one of the teen words and ten
TEEN [ten eleven twelve thirteen fourteen fifteen sixteen
      seventeen eighteen nineteen]
; subgrammar for saying 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 another one. This allows you to 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"
```

or

```
#include <filename.grammar>
```

is replaced by the contents of the file *filename.grammar*. An included file may itself contain *#include* lines. The difference between the previous two include directives is the way in which the compiler searches for the specified file, described in the following section.

Note: If you are using the Nuance command line toolkit, the use of the *#include* directive is necessary when your grammars are defined in more than one file. This is because you can only pass one file as an argument to the command line compiler.

Search and versions

The default directory location for American English grammars is *%NUANCE%\data\lang\English.America\grammars*. To cause the grammar

compiler to search for your include files locally—that is, in the directory where the compiler is run—before looking in the default directory, use the directive:

```
#include "mysubgrammars.grammar"
```

If you specify the directive:

```
#include <mysubgrammars.grammar>
```

the compiler only searches the directory

`%NUANCE%\data\lang\English.America\grammars` for the file `mysubgrammars.grammar`.

The compiler actually searches for included grammars based on the natural language of the current model set. Grammars for languages other than American English 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`. These grammars are accessible via the `#include` construct when one of the *Spanish.America* acoustic model set is used, as the model set specified to the compiler determines the specific natural language (and dialect) being used in the 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. To include the latest version of a grammar, such as the *money* grammar, use the directive:

```
#include <money.grammar>
```

If the latest version of the *money* grammar is 6.2, for example, the previous line will cause the inclusion of the file `money.grammar-v6.2`. If you want to work with a specific version of the *money* grammar, such as the version 5, you must specify the version number, as in the following directive:

```
#include <money.grammar-v5>
```

You can look at the contents of the

`%NUANCE%\data\lang\English.America\grammars` directory to see precisely which files are available.

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.

NGB and the #include directive

The use of the directive `#include` is discouraged if you are using NGB to write your grammars, but necessary otherwise. In fact, within the NGB environment, you do not need to use this directive at all because of the way grammars files are referenced through a hidden “master” grammar file.

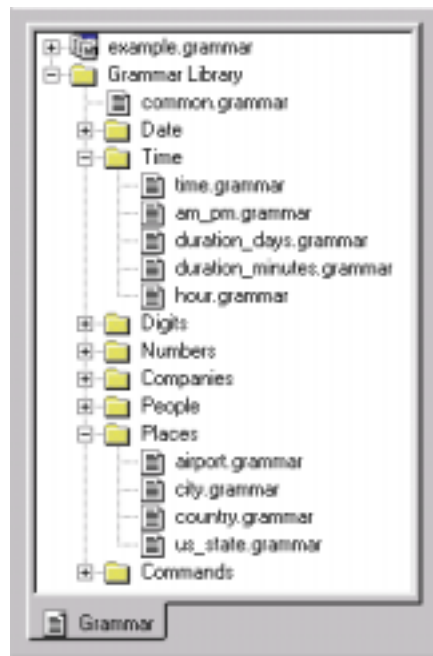
Consult the NGB help for more details on the master grammar file and how to include files in your project without explicitly using `#include`'s.

The NGB grammar library

Nuance Grammar Builder comes equipped with a Grammar Library with more than 50 subgrammars ready for use in your development. You can view the contents of a grammar file and include it in a NGB project or modify it to fit your needs.

The Grammar Library provides you with grammars most commonly used in applications such as grammars for dates, time, digits, numbers, and commands. The contents of the Grammar Library is displayed in the tree view of any NGB project.

This figure shows the NGB's project window where the Grammar Library folder is partially expanded.



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 makes reference back to *NounPhrase*.

The Grammar Specification Language 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 therefore will 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 switch. 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 40 for details on this compiler option.

Note: In NGB, set the `-dont_flatten` option by checking the Compact box in the Compiler Options dialog.

Grammar probabilities

The recognition engine uses probabilities to weight constructs while searching for matches in its data space. These weights force the recognizer to favor certain phrases over others. Typically you assign higher probabilities to phrases expected to be spoken more frequently.

Adding probabilities to your grammar can potentially increase both recognition accuracy and speed—however, assigning bad probability values can actually hurt recognition performance. Grammar probabilities are recommended only for large vocabulary grammars (over 1000 words) where probabilities can be accurately estimated based on real usage.

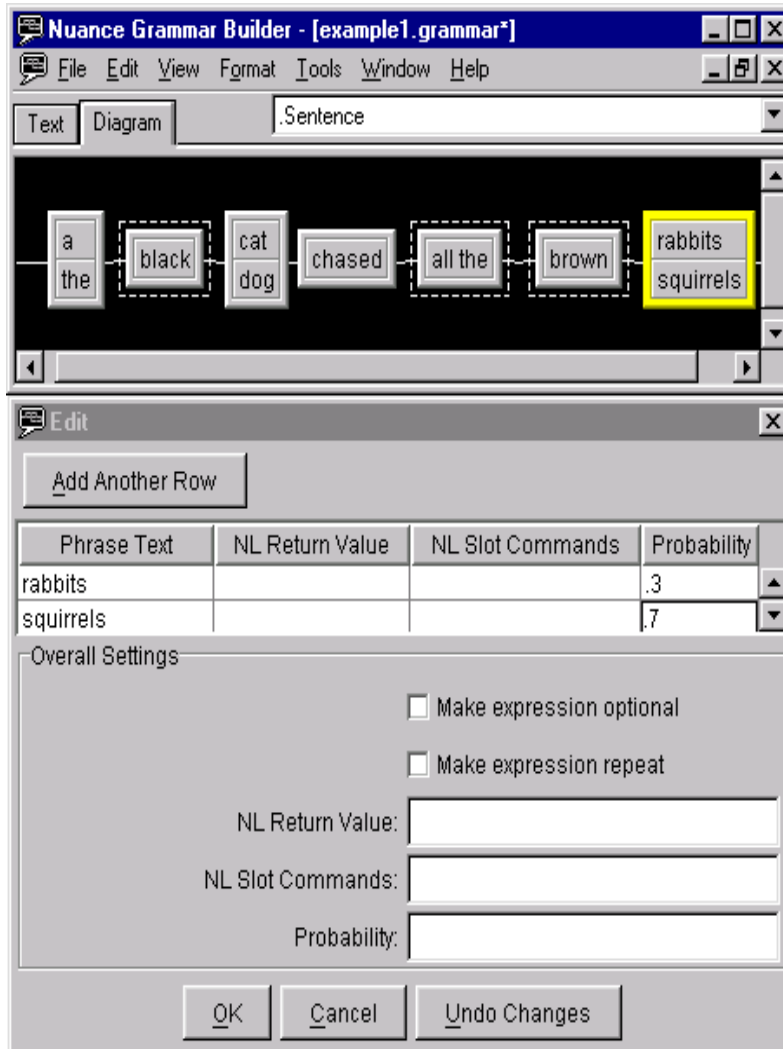
Assigning probabilities within NGB

If you are using NGB, double-clicking a box (construct) in the diagram view of a grammar brings up a dialog where you can set, among other things, an overall probability for the construct and probabilities for the phrases in that construct.

The next screen shot shows this dialog opened for the disjunct expression *[rabbits squirrels]*. The shot contains a partial display of the diagram view of the grammar *.Sentence* introduced at the beginning of this chapter.

The Edit dialog shows:

- A probability of .3 is assigned to *rabbits*
- A probability of .7 is assigned to *squirrels*
- A probability of .4 is assigned to the disjunct construct *[rabbits squirrels]*



Probabilities in GSL

You can assign probabilities to GSL constructs by using the tilde character (~). The GSL syntax to specify a probability for a construct *C* is:

C~prob

where *prob* is a non-negative number.

Probabilities are mostly used in disjunct (OR) constructs. For example, the following *City* grammar assigns different probabilities to each of the four city names:

```
City [ boston~.2
      (new_york)~.4
      dallas~.3
      topeka~.1
    ]
```

Specifying no probabilities is the same as specifying a probability of 1.0 for each item—effectively stating that any given phrase is as likely as any other.

You can also assign probabilities to the operand of optional (?), kleene closure (*), and positive closure (+) operator. The meaning of such an expression is illustrated in the following table:

Table 3: Grammar probability expressions

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%

In regards to assigning overall and individual probabilities, note that the following two probability specifications are equivalent:

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

Note: Nuance recommends that, if you choose to use grammar probabilities, you test your application’s performance both with and without probabilities. The *nuance-compile* program has the option `-dont_use_grammar_probs` that allows you to compile a package ignoring any 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 probabilities.

Dynamic grammars

A *dynamic grammar* is a grammar that can be created and modified by a running application. This section describes how to specify dynamic grammars in a GSL grammar definition, so you can add dynamic grammar support to your recognition package, and how to add dynamic grammar attachment points to your grammars so that portions of your grammars can be specified at runtime,

and changed on the fly. For more comprehensive information about dynamic grammars, see the *Nuance Application Developer's Guide*.

Dynamic grammar specification

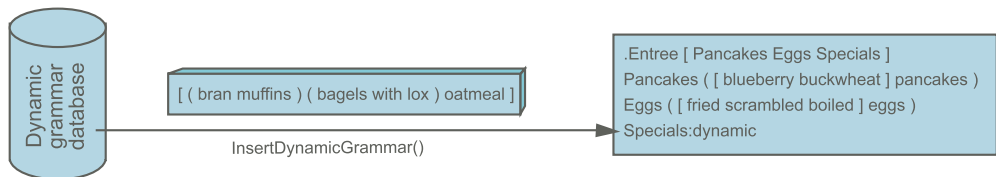
To add a dynamic grammar as a component in a top-level grammar, use the following GSL syntax:

```
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 assembles) the correct dynamic grammar in the appropriate database and inserts it into the *.Entree* grammar at the location *Specials*:



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*. Be careful, however, that any slots that are filled are defined in the target package. For information about slots see Chapter 4.

Note: 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 acoustic model set. For complete details on dynamic grammars see the *Nuance Application Developer's Guide*.

Dynamic grammar referencing

You can create static grammars that you can later reference dynamically. This can be useful if you want to create a dynamic grammar from one or more existing subgrammars. To do this, you create the static grammars using the keyword `dynaref`. For example:

```
Losangeles:dynaref [(los angeles ?california) (l a)]
    {return(la)}
Sanfrancisco:dynaref [(san francisco ?california) frisco]
    {return(sf)}
```

To compile these grammars into your application's recognition package, you must use the *nuance-compile* option `-dont_flatten`. You can later reference these in your code when building a dynamic grammar, for example:

```
NuanceStatus status = AppAddPhraseToDynamicGrammar(app, key,
    "local_cities", "[Losangeles:name Sanfrancisco:name]",
    "{<city $name>}", 1, NULL);
```

Note that referencing a grammar that does not exist in the application's recognition package causes an error—therefore, be careful in renaming or removing dynamically referenced grammars in a grammar file.

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*.

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 documentation section of the Nuance Developer Network website at extranet.nuance.com.

Compiling grammars

3

A grammar file is compiled into a recognition package using the Nuance utility *nuance-compile*. You can invoke *nuance-compile* from within Nuance Grammar Builder (NGB) or as a command-line program.

Compiling a grammar with NGB

Compiling grammars within the NGB's environment is very easy. Select the item Compile from the Tools menu—or alternatively, click the compile icon on the grammar toolbar—to display the Grammar Compiler Options dialog. You use this dialog to set the options you want to pass to the compiler and to select the master package (the set of acoustic models) to use for compilation.

The output generated by the compiler—other than the compiled package—is shown in the output window. In particular, this window displays compilation errors. One great advantage of the NGB environment is that you can go from an error description to the source of that error by double-clicking the error message.

Note: See the NGB help system for further details on how to compile grammars with NGB and how to locate the source of a compilation error.

nuance-compile inputs

When using *nuance-compile* from the command line, you must provide two arguments:

- The package name
- The name of an acoustic model set (also referred to as a master package)

For example:

```
> nuance-compile package_name model_set
```

package_name points to the location of a file named *package_name.grammar*. It can be a relative or absolute path, and identifies where *nuance-compile* looks for the grammar file and other package files. In particular, if you pass in a string without slashes, the current directory is searched. For example, the command:

```
> nuance-compile c:\home\MyApp\grammars\myapp model_set
```

tells the compiler to look for the grammar file *myapp.grammar* in the directory *c:\home\MyApp\grammars*.

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

The second required argument to *nuance-compile* is the name of an acoustic model set. Model sets (or master packages) are provided by Nuance and installed in one of the subdirectories in `%NUANCE%\data\lang`, depending on which natural language you are compiling a grammar set for. See “Choosing a model set” on page 33 for complete information on model sets.

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

Note: The subset of *nuance-compile* options discussed in this guide is listed in “Summary of compiler options” on page 42.

***nuance-compile* output**

nuance-compile generates a directory called the *package* directory. 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* argument *-o output_package_name*.

By default *nuance-compile* overwrites any previously-compiled package files. To prevent files from being overwritten, use the *nuance-compile* argument *-nooverwrite*.

The package directory 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. In particular, the file *out.compile* is always created and contains the command line used to generate the package.

Choosing a model set

The Nuance System provides multiple sets of acoustic-phonetic hidden Markov models, or *master packages*. These models 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 models 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.

Acoustic model naming convention

The default package name format for a given model is

```
language[.dialect][.model_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 French (European) French speakers is simply *French*.

The specific package name format is:

```
language[.dialect].model_type.version
```

A model that has a new package structure uses the latest `model_type` number incremented by 1, for example, *English.America.1* and *English.America.2*.

All languages have master packages with a model type number equal to 1, and this version is called the *standard* package. American English has master packages with model numbers 1, 2 and 3:

- Packages with model type number 2 are called *extended* packages. Extended packages perform additional recognition processing that may improve accuracy but requires additional memory.

- Packages with model type number 3 are called *mobile* packages and are intended for use in hands-free environments, on cellular phones, and in other environments with high levels of background noises.

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

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

Language-specific models

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

Language-specific models provided with the current release include American English, Australian-New Zealand English, British English, French, Canadian French, German, Japanese, American Spanish, and Brazilian Portuguese. Nuance releases new language-specific model sets as they become available. For a complete list of the acoustic models and languages supported, visit the NDN website at extranet.nuance.com.

Note: See the *README* file installed in `%NUANCE%\data\lang\<language>\master.package` for details on the language-specific master packages shipped with your version of the Nuance software.

Recommended usage

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

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

- *English.America* and *English.America.2* both map to the latest version of the *English.America.2* model set, for example, *English.America.2.0*.
- The model set *English.America.2* is recommended for most American English applications. This model provides fast and accurate recognition for most applications.
- If your system has memory constraints, try using *English.America.1* which uses less memory than *English.America.2*.
- 7.0 includes a new master package for American English that was developed to allow the recognition engine to perform well in hands-free environments, such as a speaker phone or a car with remote microphones for hands-free use

of a cellular phone. This model was created from a collection of acoustic models to allow for robust handling of any type of incoming speech, including normal land lines, cellular phones, and hands-free equipment. This is not currently intended to be used as the default model for all applications. Nuance is continuing to test and tune the package to verify that its performance is comparable to the current master packages for callers from non-hands-free lines. Nuance recommends that you use this master package only if your application will have a significant percentage of hands-free callers. This master package is currently only available for American English and is shipped as the model set *English.America.3*.

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 with details about the languages you want to use and the type of mixed-language support your application requires.

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 custom dictionaries within NGB

Double-click the compiler error line (in the output window) that refers to the file with the list of missing words to automatically display the file contents. You can edit that file, adding pronunciations for the missing words, and save it as a dictionary file.

Next, you add that dictionary file to your project using the File menu selection Add File to Project. The added file is displayed in the project tree as a new file icon under the Dictionary Files folder.

A recompilation will then succeed or point out phoneme errors in the dictionary file that need correction.

See the NGB help system for more information on how to deal with unknown pronunciations.

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 85. 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.2 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
```

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

Merging and overriding in NGB

Once you have added a dictionary to your project, it is very easy to specify one of the two previous alternatives. The context menu attached to a dictionary icon allows you to merge your dictionary with the master dictionary or to override it. Next time you compile your project, the merging or overriding of your dictionary file will take effect.

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.2 -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.2 -merge_dictionary
  myAltProns.dictionary
```

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

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.)

If you are using NGB, set this option by checking the Auto Pron box in the Compiler Options Dialog. Otherwise, when using the command line, recompile your package using the option `-auto_pron` on the `nuance-compile` command line as illustrated in the command:

```
> nuance-compile myGrammar English.America.2 -auto_pron
```

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

To use this within NGB, type the option followed by a file name in the Other panel of the compiler dialog box.

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.2 -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 *cross-word* 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` can use compound-word pronunciations for these types of phrases when they are specified in the dictionary.

Note: If you want to use multiword pronunciations in your package, you must not use the `nuance-compile` 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)          a j b i E m
(a t and t)      e t i * n t i

```

See “Tuning recognition performance” on page 41 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.

To minimize the size of these packages specify the compiler option *-dont_flatten*. In NGB, check the Compact box in the Grammar Compiler Dialog.

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*

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

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

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

- *The amount of search performed*

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

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

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

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

Summary of compiler options

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

Table 4: *nuance-compile* options

Option	Notes
-merge_dictionary <i>filename</i>	Merges standard dictionary entries with custom dictionary entries.

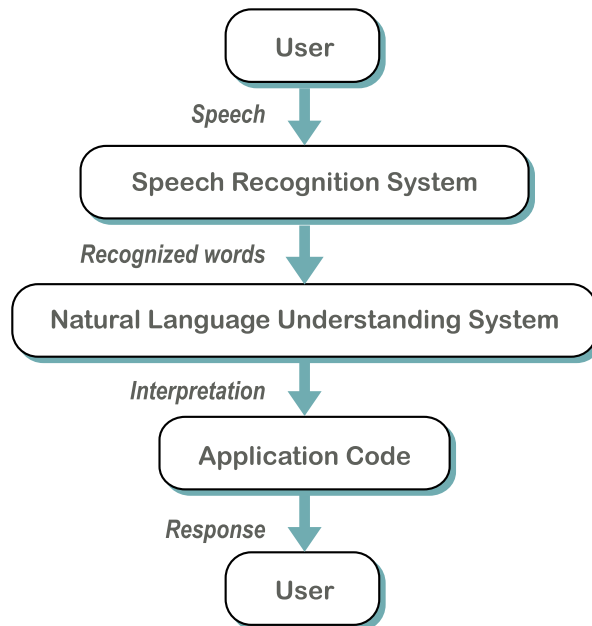
Table 4: nuance-compile options

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

Natural language understanding

4

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



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

“Withdraw fifteen hundred dollars from savings”

“Take fifteen hundred out of savings”

“Give me one thousand five hundred dollars from my savings account”

map to the same semantic units:

action = withdraw

amount = 1500.00

account = savings

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

Defining natural language interpretations

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

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

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

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

command-type = transfer

source-account = savings

destination-account = checking

amount = 500

Natural language commands

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

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

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

while in the next one, the command attaches to the entire OR construction,

```
[checking savings]:
```

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

There are two kinds of natural language commands:

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

Slot-filling commands

A slot-filling command has the form:

```
<Slot Value>
```

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

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

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

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

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

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

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

you should create a grammar like the following:

```

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

```

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

```

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

```

Return commands and variables

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

```

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

```

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

Assigning variables

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

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

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

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

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

Caution: If you omit the ‘\$’ before the variable name, the slot is filled with the string “acct” rather than the value of the variable.

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

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

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

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

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

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

```
{ return(cat) }
{ return("cat") }
```

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

```
{ return(1) }
{ return("1") }
```

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

The *string* variable

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

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

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

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

How slots are filled

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

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

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

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

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

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

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

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

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

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

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

- *Commands have precedence over the unary operators*

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

Consider the following grammar:

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

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

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

The slot definitions file

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

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

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

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

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

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

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

Compiling a grammar with natural language support

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

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

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

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

Ambiguous grammars

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

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

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

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

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

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

- Program logic
- A subdialog

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

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

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

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

Advanced features

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

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

Functions

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

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

Standard functions

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

```
NonZeroDigit [ one      {return(1)}
               two      {return(2)}
               three    {return(3)}
               four     {return(4)}
               five     {return(5)}
               six      {return(6)}
               seven    {return(7)}
               eight    {return(8)}
               nine     {return(9)}
             ]

TwentyToNinety [ twenty   {return(20)}
                 thirty   {return(30)}
                 forty    {return(40)}
                 fifty    {return(50)}
                 sixty    {return(60)}
                 seventy  {return(70)}
                 eighty   {return(80)}
                 ninety   {return(90)}
             ]

TwoDigit [ ( TwentyToNinety:num1 NonZeroDigit:num2 )
           {return(add($num1 $num2))}
           TwentyToNinety:num1
           {return($num1)}
         ]
```

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

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

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

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

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

Table 5: Standard integer and string functions

Function	Description	Unset variable
add	Returns the sum of two integers.	0
sub	Returns the result of subtracting the second integer from the first.	0
mul	Returns the product of two integers.	1
div	Returns the truncated integer result of dividing the first integer by the second (e.g., <code>div(9 5)</code> 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 <code>strcat(\$a1 strcat(\$a2 \$a3))</code> , which effectively concatenates three strings.	"" (empty sting)

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

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

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

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

User-defined functions

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

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

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

```

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

```

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

```

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

```

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

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

```

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

```

Complex values

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

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

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

Structures

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

```

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

```

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

```

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

```

```

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

```

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

Referencing feature values

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

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

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

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

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

```

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

```

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

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

```

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

```

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

Nested structures

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

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

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

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

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

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

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

Lists

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

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

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

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

List functions

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

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

```

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

```

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

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

Table 6: Built-in list functions

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

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

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

List commands

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

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

To use the value returned by this function, it must be used within a command:

```
{return (concat($list1 $list2))}
```

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

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

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

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

This table lists the supported list commands:

Table 7: List commands

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

These commands modify the value of the first argument. So the command `insert-begin(list1 2)`

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

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

Testing grammars

5

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

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

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

Grammar testing overview

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

Coverage test

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

Interpretation test

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

Over-generation test

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

Ambiguity test

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

Pronunciation test

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

Testing a grammar with Nuance Grammar Builder

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

Creating an initial test set

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

- *Have the prompt wording in mind.*

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

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

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

- san jose
- my departure city is san jose
- it is san jose
- it's san jose
- uh it is san jose
- the departure city is san jose
- san jose california
- *Do not generate your initial test set directly from the grammar.*

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

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

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

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

Step-by-step grammar testing instructions

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

Open the project

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

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

Compile the grammar

- 1 From the Tools menu, select the Compile command.

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

- 2 Select *English.America.1* from the pull down list in the Master Package box.
- 3 Leave the Auto Pron box unchecked. This prevents the compiler from performing automatic pronunciation generation.
- 4 Click OK to start the compilation.

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

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

Define the missing slot

- 1 Open the project file *tutorial_master.slot_definitions* by double-clicking its icon in the project tree view.
- 2 Follow the directions in the file to fix the bug—uncomment the line that contains the word `origin`.
- 3 Save the slot definitions file.

Recompile and find the second error

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

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

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

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

- 2 Double-click on the error message to view the *tutorial_master.missing* file. Notice that it contains two words “columnbus” (a misspelling of the word “columbus”) and “warshington” (a misspelling of “washington”).
- 3 Open the *departure_city.grammar* file.
- 4 Locate the word “columnbus” (try searching for it) in that grammar file.
- 5 Change the words “columnbus” to “columbus” and “warshington” to “washington”, and save the changes.
- 6 Recompile with the shown options—identical to those used for the previous compilation.

This time the grammar should compile with no errors.

Open the test set

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

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

Coverage test

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

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

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

Notice that the phrase “we are leaving from las vegas” has the corresponding entry “<Out of Grammar>” in the Current Result(s) column. The reason is that “las vegas” is missing from the grammar. To fix this problem, add “las vegas” to the grammar.

- 3 Open the grammar file by double-clicking on *departure_city.grammar* in the project window.
- 4 Look for the definition of the City subgrammar, and insert the string “(las vegas) {return(“las_vegas”)}” in between “washington” and the closing square-bracket “].”
- 5 Save the changes and recompile the project.
- 6 Rerun the command Interpret All Sentences. Compare the entries in the Saved Results and the Current Results columns for the phrase “we are leaving from las vegas” (the last one on the list). Notice that the Current Results column now contains the interpretation “las_vegas” in the origin slot.

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

NL interpretation test

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

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

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

- 2 Open the grammar file, find the bug, and fix it.
- 3 Recompile, and rerun the test.

Ambiguity test

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

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

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

- 1 Select Generate Sentences from the Tools menu.

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

- 2 In the Generate Options dialog:
 - Enter 200 as the number of sentences to generate
 - Check the NL box and the Ambiguous Only box
 - Click OK

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

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

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

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

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

Over-generation test

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

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

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

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

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

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

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

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

- 3 Open the grammar file, and try to find where the word “please” occurs in the definition of `.DEPARTURE_CITY`. Try to change the grammar so that “please” only occurs where it sounds natural.
- 4 Recompile and retest to see if you have solved the problem.

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

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

Regression tests

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

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

Command-line tools for grammar testing

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

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

- *Xapp*
- *batchrec*

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

parse-tool

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

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

produces the following output:

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

generate

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

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

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

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

Then the output might look like:

```
eighty one
seven one
fifty
```

```
fourteen
thirty two
```

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

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

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

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

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

nl-tool

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

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

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

Xapp

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

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

To run *Xapp*:

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

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

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

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

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

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

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

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

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

3 Select the grammar you want to test.

4 Cycle through the following steps until you are done:

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

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

batchrec

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

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

Testing recognition performance

6

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

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

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

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

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

Choosing *batchrec* test sets

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

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

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

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

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

Using *batchrec*

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

- The recognition package

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

For example:

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

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

Creating the testset file

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

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

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

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

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

Additional commands

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

```
*Echo text
```

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

```
*Exit
```

Lets you explicitly exit the testset at a given point.

`*SetParam param_name=param_value`

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

`*GetParam param_name`

Prints out the current value of the specified parameter.

`*NewAudioChannel`

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

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

Dynamic grammar commands

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

`*OpenDynamicGrammarDatabase db_handle DBDescriptor_arguments`

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

`*CloseDynamicGrammarDatabase db_handle`

Closes the connection to the specified database.

`*NewDynamicGrammarEmpty db_handle db_key ok_to_overwrite`

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

`*NewDynamicGrammarFromGSL db_handle db_key gsl_file ok_to_overwrite`

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

- *NewDynamicGrammarFromPhraseList db_handle db_key pl_file ok_to_overwrite*
- Creates a new record in a dynamic grammar database containing the specified set of phrases. The phrase list file (*pl_file*) contains one phrase per line, in the format:
- phrase_id phrase_text nl_command probability*
- For example:
- ```
user_112 "john doe" "{<user john_smith>}" 1
```
- Note that the natural language command *must* be enclosed in double quotes.
- \*DeleteDynamicGrammar db\_handle db\_key*
- Removes a record from a dynamic grammar database.
- \*CopyDynamicGrammar source\_db\_handle source\_db\_key target\_db\_handle target\_db\_key ok\_to\_overwrite*
- Creates a new record in a dynamic grammar database, containing the contents of an existing record.
- \*AddPhraseToDynamicGrammar db\_handle db\_key p\_id p\_text p\_nl*
- \*AddPhraseList db\_handle db\_key pl\_file*
- Adds a single phrase or a set of phrases to a dynamic grammar.
- \*RemovePhrase db\_handle db\_key p\_id*
- Removes a phrase from a dynamic grammar.
- \*QueryDynamicGrammarExists db\_handle db\_key*
- Prints out whether the given database contains a dynamic grammar at a given key.
- \*QueryDynamicGrammarContents db\_handle db\_key*
- \*QueryDynamicGrammarContentsWithID db\_handle db\_key phrase\_id*
- Prints out the contents of a specified dynamic grammar database record, or of the phrases in that record with a given ID.
- \*InsertDynamicGrammar db\_handle db\_key label {CALL | PERMANENT}*
- Inserts a dynamic grammar from a database into a static grammar at the given label, with the specified persistence (*CALL* or *PERMANENT*). *CALL* persistence means that the insertion remains until next `NewAudioChannel` line.

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

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

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

`*EnrollCommitPhrase`

Commits an updated dynamic grammar to the database.

`*EnrollAbortPhrase`

Ends an enrollment session without updating the dynamic grammar.

`*ModifyPhrase db_handle db_key p_id p_text p_nl`

Modifies a voice-enrolled phrase in a dynamic grammar.

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

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

## Optional arguments

*batchrec* supports a number of optional arguments:

`-transcriptions transcriptions_file`

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

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

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

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

`-nl_transcriptions` *nl\_transcriptions\_file*

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

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

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

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

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

rather than like this:

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

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

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

`-grammar` *grammar\_name*

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

`-vrs`

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

`-rcengine`

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

`-rcapi`

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

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

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

`-rc_timeout` *float*

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

`-app`

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

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

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

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

`-endpoint` | `-dont_endpoint`

The `-endpoint` and `-dont_endpoint` options specify whether or not endpointing is applied to the speech samples before recognition.

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

`-print_confidence_scores`

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

`-print_word_confidence_scores`

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

`-debug_level` 0-5

`-debug_level_during_init` 0-5

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

## Setting Nuance parameters

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

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

causes *batchrec* to perform N-best processing.

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

## Output from *batchrec*

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

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

This output includes the following information:

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

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

```
Grammar: .Sentence
```

Names the recognition grammar used to recognize this file.

```
Transcription: buy him seven fish now
```

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

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

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

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

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

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

Prints the natural language interpretation generated from the recognition result.

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

Total Audio: 2.63 sec

Provides the length of the recording.

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

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

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

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

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

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

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

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

NL Status: correct

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

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

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

# Creating application-specific dictionaries

# 7

---

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

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

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

## The Nuance phoneme set

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

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

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

**Table 8: Mappings for American English**

| Phonetic Category | Phoneme | Example                                  | Phoneme | Example                  |
|-------------------|---------|------------------------------------------|---------|--------------------------|
| Vowels            | i       | fleet                                    | u       | blue                     |
|                   | I       | dimple                                   | U       | book                     |
|                   | e       | date                                     | o       | show                     |
|                   | E       | bet                                      | O       | caught <sup>a</sup>      |
|                   | a       | cat                                      | A       | father, cot <sup>a</sup> |
|                   | aj      | side                                     | aw      | couch                    |
|                   | Oj      | toy                                      | *r      | bird                     |
|                   | ^       | cut                                      | *       | alive                    |
| Stops             | p       | put                                      | b       | ball                     |
|                   | t       | take                                     | d       | dice                     |
|                   | k       | catch                                    | g       | gate                     |
| Flap              | !       | but <del>t</del> er, hid <del>d</del> en |         |                          |
| Nasals            | m       | mile                                     | g~      | runni <del>ng</del>      |
|                   | n       | nap                                      |         |                          |

**Table 8: Mappings for American English**

| Phonetic Category                    | Phoneme    | Example       | Phoneme    | Example           |
|--------------------------------------|------------|---------------|------------|-------------------|
| Fricatives                           | <b>f</b>   | <i>friend</i> | <b>v</b>   | <i>voice</i>      |
|                                      | <b>T</b>   | <i>path</i>   | <b>D</b>   | <i>them</i>       |
|                                      | <b>s</b>   | <i>sit</i>    | <b>z</b>   | <i>zebra</i>      |
|                                      | <b>S</b>   | <i>shield</i> | <b>Z</b>   | <i>vision</i>     |
|                                      | <b>h</b>   | <i>have</i>   |            |                   |
| Affricates                           | <b>tS</b>  | <i>church</i> | <b>dZ</b>  | <i>judge</i>      |
| Approximants                         | <b>j</b>   | <i>yes</i>    | <b>w</b>   | <i>win, which</i> |
|                                      | <b>r</b>   | <i>row</i>    | <b>l</b>   | <i>lame</i>       |
| Others<br>(reserved for<br>Compiler) | -          | (silence)     |            |                   |
|                                      | <b>inh</b> | (inhale)      | <b>exh</b> | (exhale)          |
|                                      | <b>clk</b> | (click)       | <b>rej</b> | (other)           |

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

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

**Table 9: Mapping English-language letters**

| <b>Letter</b> | <b>Phonemes</b> | <b>Example</b>                          |
|---------------|-----------------|-----------------------------------------|
| <i>a</i>      | <b>e</b>        | <b>able</b>                             |
|               | <b>A</b>        | <b>father</b>                           |
|               | <b>a</b>        | <b>apple, pan</b>                       |
|               | <b>O</b>        | <b>ball, caught, pawn</b>               |
|               | <b>*</b>        | <b>alive, zebra (unstressed vowels)</b> |
| <i>b</i>      | <b>b</b>        | <b>ball</b>                             |
| <i>c</i>      | <b>k</b>        | <b>cat, echo</b>                        |
|               | <b>tS</b>       | <b>child</b>                            |
| <i>d</i>      | <b>d</b>        | <b>dogs</b>                             |
|               | <b>!</b>        | <b>bridle</b>                           |
|               | <b>t</b>        | <b>asked, kissed</b>                    |
| <i>e</i>      | <b>i</b>        | <b>even</b>                             |
|               | <b>E</b>        | <b>echo</b>                             |
|               | <b>u</b>        | <b>new</b>                              |
|               | <b>*</b>        | <b>agent</b>                            |
| <i>f</i>      | <b>f</b>        | <b>father</b>                           |
| <i>g</i>      | <b>g</b>        | <b>golf</b>                             |
|               | <b>dZ</b>       | <b>giant</b>                            |
| <i>h</i>      | <b>h</b>        | <b>hotel</b>                            |
| <i>i</i>      | <b>aj</b>       | <b>ivory, time</b>                      |
|               | <b>l</b>        | <b>India</b>                            |
|               | <b>i</b>        | <b>Lisa, shield</b>                     |
|               | <b>*</b>        | <b>sanity, aphid</b>                    |
| <i>j</i>      | <b>dZ</b>       | <b>jump</b>                             |
| <i>k</i>      | <b>k</b>        | <b>key</b>                              |

**Table 9: Mapping English-language letters**

| Letter     | Phonemes   | Example                     |
|------------|------------|-----------------------------|
| <i>l</i>   | <b>l</b>   | <i>law</i>                  |
|            | <b>* l</b> | <i>able</i>                 |
| <i>m</i>   | <b>m</b>   | <i>Mike, thumb</i>          |
|            | <b>* m</b> | <i>Adam, wisdom</i>         |
| <i>n</i>   | <b>n</b>   | <i>new, knot</i>            |
|            | <b>* n</b> | <i>widen, nation</i>        |
| <i>n+g</i> | <b>g~</b>  | <i>ring</i>                 |
| <i>o</i>   | <b>o</b>   | <i>open, coat, show</i>     |
|            | <b>A</b>   | <i>olive</i>                |
|            | <b>aw</b>  | <i>cow</i>                  |
|            | <b>O</b>   | <i>bought</i>               |
|            | <b>u</b>   | <i>shoe, loop</i>           |
|            | <b>U</b>   | <i>book, would</i>          |
|            | <b>^</b>   | <i>company</i>              |
|            | <b>*</b>   | <i>carrot</i>               |
| <i>p</i>   | <b>Oj</b>  | <i>toy</i>                  |
|            | <b>p</b>   | <i>pull</i>                 |
| <i>q</i>   | <b>f</b>   | <i>aphid</i>                |
|            | <b>k w</b> | <i>quick</i>                |
| <i>r</i>   | <b>k</b>   | <i>Iraq</i>                 |
|            | <b>r</b>   | <i>ring</i>                 |
| <i>s</i>   | <b>*r</b>  | <i>bird, rider, grammar</i> |
|            | <b>s</b>   | <i>soup</i>                 |
|            | <b>S</b>   | <i>shell</i>                |
|            | <b>z</b>   | <i>dogs, wisdom</i>         |
|            | <b>Z</b>   | <i>vision</i>               |

**Table 9: Mapping English-language letters**

| Letter   | Phonemes   | Example                       |
|----------|------------|-------------------------------|
| <i>t</i> | <b>t</b>   | <i>time</i>                   |
|          | <b>!</b>   | <i>butter</i> , <i>sanity</i> |
|          | <b>D</b>   | <i>them</i>                   |
|          | <b>T</b>   | <i>thin</i>                   |
|          | <b>tS</b>  | <i>question</i>               |
| <i>u</i> | <b>S</b>   | <i>nation</i>                 |
|          | <b>^</b>   | <i>under</i> , <i>putt</i>    |
|          | <b>U</b>   | <i>put</i> , <i>would</i>     |
|          | <b>u</b>   | <i>lunar</i> , <i>rude</i>    |
|          | <b>ju</b>  | <i>usage</i> , <i>confuse</i> |
| <i>v</i> | <b>*</b>   | <i>focus</i>                  |
|          | <b>v</b>   | <i>very</i>                   |
| <i>w</i> | <b>w</b>   | <i>wisdom</i> , <i>which</i>  |
| <i>x</i> | <b>z</b>   | <i>xenophobia</i>             |
|          | <b>k s</b> | <i>axis</i>                   |
| <i>y</i> | <b>j</b>   | <i>yes</i>                    |
|          | <b>i</b>   | <i>very</i>                   |
|          | <b>aj</b>  | <i>cry</i>                    |
| <i>z</i> | <b>z</b>   | <i>zebra</i>                  |

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 t S |
|          | s a n w I t S   |

**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 a j 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 a j l d
coat k o t
compass k ^ m p * s
confuse k * n f j u z
cow k a w
cry k r a j
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 n~         |
| 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 a j 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 a j ! * 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.

## 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 American English, Nuance supports a variety of other languages and English dialects, including Australian and U.K. English, French, German, Japanese, Brazilian Portuguese, and Latin American Spanish.

To work with any language, you need to use a specific Nuance model set to process that language. See the section “Choosing a model set” on page 33. The *nuance-master-packages* utility will also output a list of the models available on your system.

Tables for mapping the phonemes for all supported languages to their CPA representation are located at the documentation site on the Nuance Developer Network website at [extranet.nuance.com](http://extranet.nuance.com).

# GSL reference



This appendix provides a 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”      | The literal string xyz should actually be typed into the grammar file |
| 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 lists format definitions of the GSL syntax rules:

```
GrammarFile ::= GrammarDefinitions
GrammarDefinitions ::= GrammarDefinition |
 GrammarDefinition GrammarDefinitions
GrammarDefinition ::= GrammarName GrammarConstruct |
 GrammarName~":dynamic"GrammarConstruct |
 GrammarName~":dynaref"GrammarConstruct |
 GrammarName~":filler"GrammarConstruct
GrammarConstruct ::= BareConstruct | BareConstruct Commands
BareConstruct ::= BareConstructWithoutProb |
 BareConstructWithoutProb~"~"~Prob
```

```

BareConstructWithoutProb ::= Word
 GrammarReference
 "?" GrammarConstruct
 "*" GrammarConstruct
 "+" GrammarConstruct
 "[" GrammarConstructs "]"
 "(" GrammarConstructs ")"
GrammarConstructs ::= GrammarConstruct |
 GrammarConstruct GrammarConstructs
Word ::= WordChar | WordChar~Word
GrammarReference ::= GrammarName | GrammarName~":"~Variable
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*

## 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 11: 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                                             |



# Index

---

## SYMBOLS

#include 21, 22

## A

acoustic models  
    naming conventions 33  
    usage 34  
acoustic-phonetic models 10  
add operator 54  
adding NL commands 9  
adding probabilities 25  
AddPhraseList 77  
AddPhraseToDynamicGrammar 77  
ambiguity test 62, 66  
anticipating responses 6  
-app option to *batchrec* 80  
*arpabet-to-cpa* 94  
audio file 75  
-auto\_pron option to *nuance-compile* 38

## B

*batchrec* 73–84  
    commands 76  
    Nuance parameters 81  
    optional arguments 78  
    output 82  
    testset example 75  
    timeout 80  
    usage 74  
*batchrec* commands  
    \*Echo 75  
    \*Exit 75  
    \*GetParam 76  
    \*NewAudioChannel 76  
    \*SetParam 76

## C

cellular phones 35  
CITY grammar 8

CloseDynamicGrammarDatabase 76  
commands, natural language 46  
    list 58  
    return 48  
    slot-filling 47  
-compare option to *nl-tool* 70  
CompileAndInsertDynamicGrammar 78  
compiling recognition packages  
    *see nuance-compile*  
compound-word modeling 38, 39  
Computer Phonetic Alphabet 86  
    *see* CPA  
CopyDynamicGrammar 77  
coverage test 61, 65  
CPA 86–91  
    converting to 94  
crossword modeling 38  
crosswords 41

## D

DATE grammar 8  
-debug\_level option to *batchrec* 81  
defining slots 4  
DeleteDynamicGrammar 77  
dialog  
    directed 5  
    mixed-initiative 5  
dialog design 3  
dictionaries  
    creating 85–94  
    sample 92  
dictionary  
    example 38  
    merge 37  
    override 37  
    standard 35  
dictionary example 92  
directed dialog 5  
div operator 54  
-do\_crossword option to *nuance-compile* 41  
-dont\_endpoint option to *batchrec* 81  
-dont\_flatten option to *nuance-compile* 29

- dynamic grammar 27
  - dynaref keyword 29
  - enrollment 29
  - GSL support 28
  - referencing 29

## E

- Echo, *see batchrec* commands
- endpoint option to *batchrec* 81
- EnrollAbortPhrase 78
- EnrollCommitPhrase 78
- enrollment 29
- EnrollNewPhrase 78
- Exit, *see batchrec* commands

## F

- features 56
- filling multiple slots 48
- flight info example 4, 5
- functions 53–55
  - standard 53
  - user-defined 54

## G

- gen\_ref option to *nl-tool* 70
- generate* 69–70
- generate-nlref* 79
- GetParam, *see batchrec* commands
- grammar
  - core 7
  - definition 16
  - description 16, 17
  - design principles 1
  - dynamic 27
  - example 13
  - example of naturally phrased numbers 21
  - filler 7, 8
  - flattening 40
  - name 16, 17
  - operators 18
  - probabilities 25

- prompt coordination 3
- recursion 24
- subgrammars 20
- testing 63
- top-level 20
- grammar compiler
  - see nuance-compile*
- grammar development 1
  - designing responses 2
  - guidelines 1
  - predicting responses 2
  - tasks 3
- grammar file 16
  - comments 18
  - compound words 38
  - dynamic grammar support 28
  - dynamic keyword 28
  - dynaref keyword 29
  - enrollment support 29
  - grammar naming conventions 17
  - including other files 21
  - lists 57
  - reserved strings 19
  - sample 21
- Grammar keyword 75
- grammar option to *batchrec* 75, 80
- Grammar Specification Language
  - See GSL*
- grammar writing recommendations 30
- GSL 9
  - code example 9
  - enrollment support 29
  - include directive 22
  - probabilities 25
  - reserved strings 97
  - syntax 95
- GSL operators
  - concatenation 18
  - disjunction 18
  - kleene closure 18
  - optional 18
  - positive closure 18

## I

- include directive 21

- InsertDynamicGrammar 77
  - integer functions, in natural language
    - slots 54
  - International Phonetic Alphabet 86
  - interpretation test 61, 66
- L**
- language-specific conventions 30
  - lists, as interpretation values 57
    - commands 58
    - functions 57
- M**
- master packages 33
  - merge\_dictionary option to
    - nuance-compile* 37
  - misspellings 37
  - mixed-initiative dialog 5
  - ModifyPhrase 78
  - mul operator 54
  - multiple pronunciations 91
  - multiword 38
  - multiword modeling 38
  - multiwords and *-dont\_flatten* option 38
- N**
- natural language commands 9
  - natural language interpretation 45–59
    - commands 46
    - compiling packages 51
    - complex slot values 55
    - features 56
    - lists 57
    - return commands 48
    - slot commands 47
    - variables 48
  - N-best processing 79
  - neg operator 54
  - nested structures 57
  - NewAudioChannel, *see batchrec*
    - commands
  - NewDynamicGrammarEmpty 76
  - NewDynamicGrammarFromGSL 76
  - NewDynamicGrammarFromPhraseList
    - 77
  - NGB 13
    - assigning probabilities 25
    - editing grammar attributes 15
    - grammar diagram view 14, 19
    - Grammar Library 8, 23
    - grammar text view 19
    - grammar writing support 19
    - include directive 23
  - NL commands and unary operators 50
  - NL transcriptions file 79
  - nl\_transcriptions option to *batchrec* 79
  - nl-compile* 51
  - nl-tool* 70
  - Nuance Developer Network x
  - Nuance phoneme set 86
  - nuance-compile* 10, 31
  - nuance-master-packages* 34
  - num option to *generate* 69
- O**
- OpenDynamicGrammarDatabase 76
  - operators
    - add 54
    - div 54
    - mul 54
    - neg 54
    - strcat 54
    - sub 54
  - over-generation test 62, 67
  - override\_dictionary option to
    - nuance-compile* 37
- P**
- package 31–32
  - package directory 32
  - package files summary 97
  - package name convention 33
  - package optimization 39–41
    - graph algorithms 40
    - size 40

- unflattened grammars 40
- parse-tool* 69
- phonemes 85–91
  - American English 86
  - mappings to letters 87
- phrase pronunciation 38
- principles of grammar development 1
- print\_confidence\_scores option to
  - batchrec* 81
- print\_trees option to *parse-tool* 69
- print\_word\_confidence\_scores option
  - to *batchrec* 81
- probabilities and
  - expressions 25
  - kleene closure 27
  - optional operator 27
  - positive closure 27
- prompt design 5
- prompt wording 62
- pronounce* 36
- pronunciation test 62
- pronunciations 35–39
  - compound-word 38
  - cross-word 38
  - missing words 37
  - multiple 91
  - multiword 38

## Q

- QueryDynamicGrammarContents 77
- QueryDynamicGrammarContentsWithI
  - D 77
- QueryDynamicGrammarExists 77

## R

- rc\_timeout option to *batchrec* 80
- rcapi option to *batchrec* 80
- rcengine option to *batchrec* 80
- recognition
  - tuning, with *batchrec* 73
- recognition package 10
  - slot definitions files 50
- recognition server 71

- recording
  - with *batchrec* 73
- recursive grammars 24
- referencing variables 50
- regression test 68
- RemovePhrase 77
- return command 48

## S

- server 71
- SetParam *see batchrec* commands
- slot definition 4
- slot filling and values 50
- slot name 79
- slot-filling commands 47–50
- slots
  - complex values 55
  - definitions file 50
  - features 56
  - nested structures 57
  - structures 55
- standard dictionary 35
- strcat operator 54
- string functions, in natural language
  - slots 54
- string variable 49
- structures 56
- sub operator 54
- subgrammar 19, 20
- summary of package files 97

## T

- technical support x
- test set 62
- testing
  - ambiguity 62
  - command-line programs 68
  - coverage 61
  - interpretation 61
  - over-generation 62
  - pronunciation 62
  - step-by-step 63–68
- testset option to *batchrec* 75

transcriptions file 78  
  example 79  
-transcriptions option to *batchrec* 78

## U

unary operators and NL commands 50  
user-defined functions 54

## V

variables 50  
  assigning 48

  valid characters 49  
-vrs option to *batchrec* 80

## W

*wavconvert* 73  
word names 17  
-write\_auto\_pron\_output option to  
  *nuance-compile* 38

## X

*Xwavedit* 73

