

Nuance Speech Recognition System
Version 7.0

Grammar Developer's Guide

Petitioner has added romanettes to pages (i) - (ii). Otherwise, it leaves the original page numbering.

MICROSOFT CORP.
EXHIBIT 1012

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
nuance-compile inputs 31
nuance-compile 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
parse-tool 69
generate 69
nl-tool 70
Xapp 70
batchrec 71

Chapter 6. Testing recognition performance 73
Choosing *batchrec* test sets 74
Using *batchrec* 74
Creating the testset file 75
Optional arguments 78
Setting Nuance parameters 81
Output from *batchrec* 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:

italic	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*.

Chapter 1

The grammar development process

1

An intuitive user interface that guides the user through a constrained but purposeful interaction is a crucial component of speech recognition application design. The user speaks naturally, yet, 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.

What city would you like to leave from?	
San Francisco	[the city name by itself]
I'd like to leave from San Francisco	[a literal response]
Uh, San Francisco	[initial hesitation]
San Francisco, please	[final "please"]
(I'm) leaving from San Francisco	
(I'm) departing from San Francisco	[some additional possibilities]
What city would you like to fly to?	
New York	[the city name by itself]
I'm flying to New York	[a literal response]
I'd like to fly to New York	[another literal response]
Uh, New York	[initial hesitation]
New York, please	[final "please"]
(I'm) going to New York	[some additional possibilities]
My destination is New York	
What date would you like to leave?	
May second	[the date by itself]
I'd like to leave on May second	[a literal response]
I'm leaving on May second	[a second literal response]
Leaving May second	[a third literal response]
Um, May second, please	[hesitation + final "please"]

What time would you like to depart?	
2 pm	[the time by itself]
I'd like to depart at 2 pm	[a literal response]
I'm departing at 2 pm	[a second literal response]
Departing 2 pm	[a third literal response]
2pm, please	[final "please"]
You're going from <origin> to <dest> on <date> at <time>. Is this correct?	
Yes	["yes" by itself]
No	["no" by itself]
Yes, that's correct	[a literal response]
Yes it is	[a second literal response]
No, that's not correct	[a third literal response]
No, it's not	[a fourth literal response]
Yeah (or yup, or you bet)	[casual alternatives]
Would you like to start over or hang up?	
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>}
```

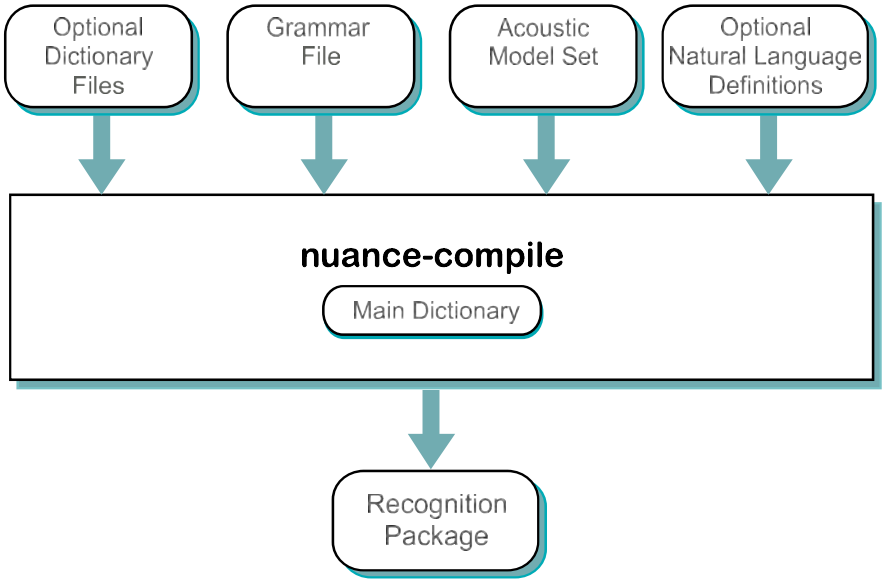
The grammars are now ready to be compiled and tested.

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.

Chapter 2

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

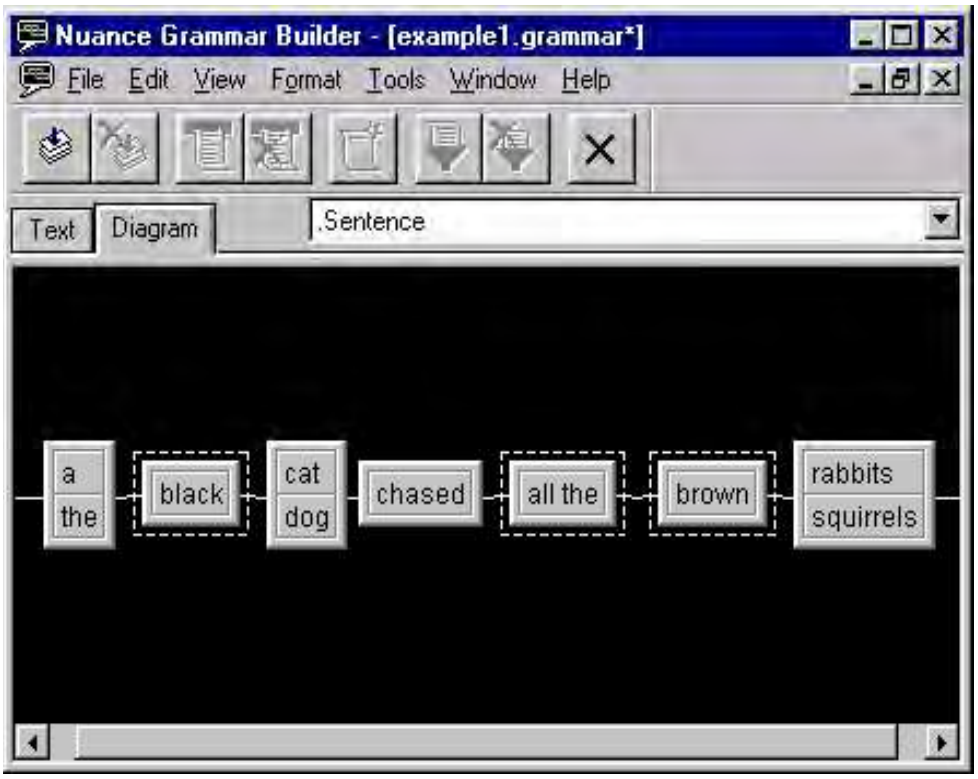
You specify a grammar in a *grammar file*. A grammar file is a text file—it can contain more than one grammar definition, and it has the file extension *.grammar*, for example, *myApp.grammar*. A grammar file is the basic component necessary to build a recognition package.

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

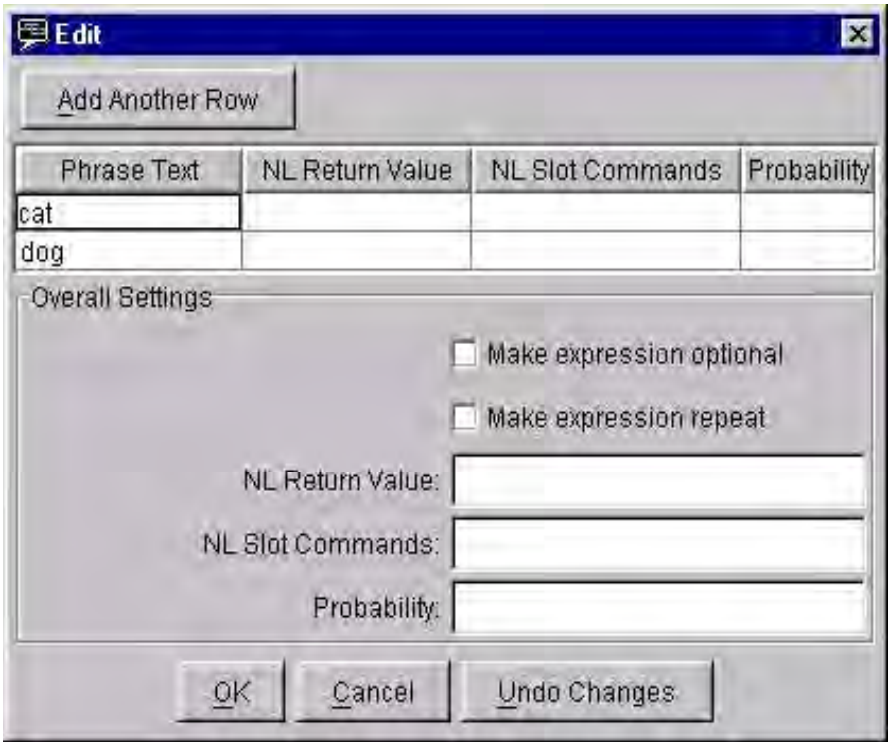
Diagram tab of the window shows the graphical representation of a grammar named *.Sentence*.



Each box in the diagram represents a grammar construct:

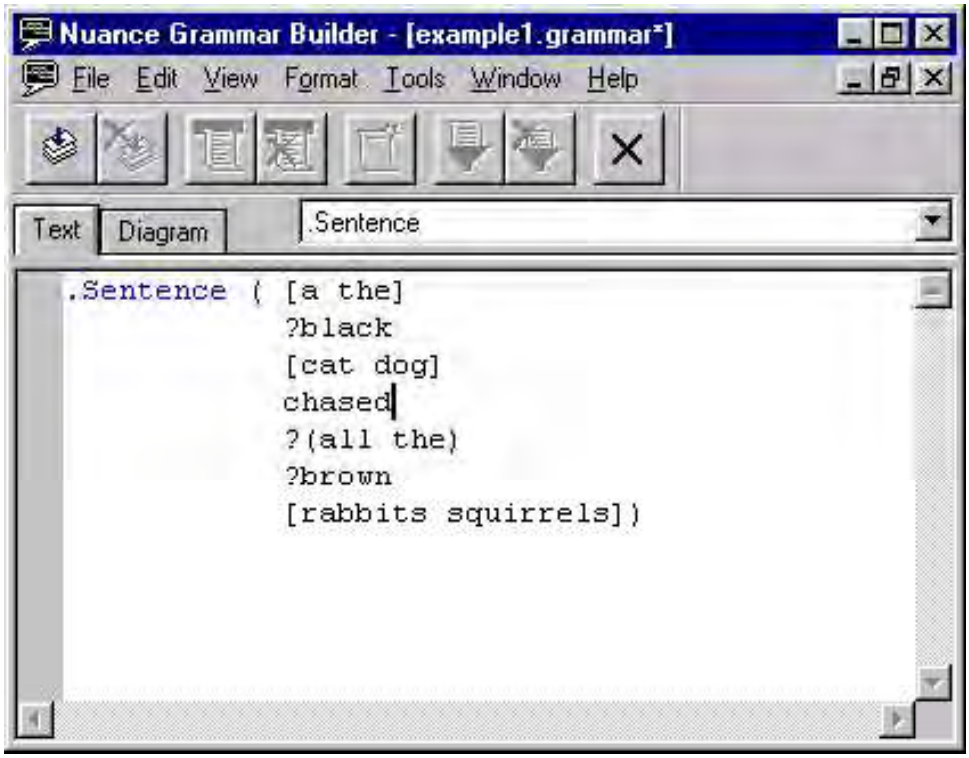
- Optional constructs—like *black*—are enclosed in a dotted white box
- Sequential constructs—like the components of one whole sentence—are linked from left to right
- Alternate constructs—like *cat/dog*—are represented vertically stacked

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

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

`[morning afternoon evening]`

"morning", "afternoon", "evening"

`(good [morning afternoon evening])`

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

`(?good [morning afternoon evening])`

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

`(thanks +very much)`

"thanks very much", "thanks very very much", and so on

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

"thanks much", "thanks very much", "thanks very very much", and so on

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

`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

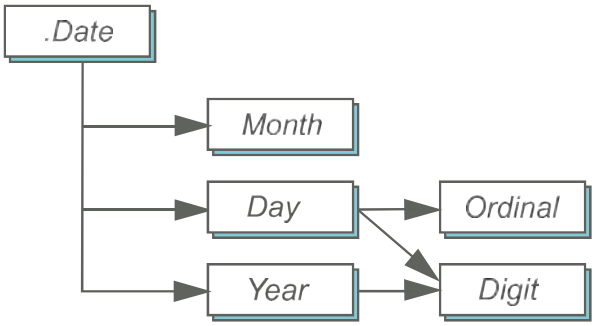
- Helps focus the grammar development to the task at hand
- Hides unnecessary details and promotes modularity

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.

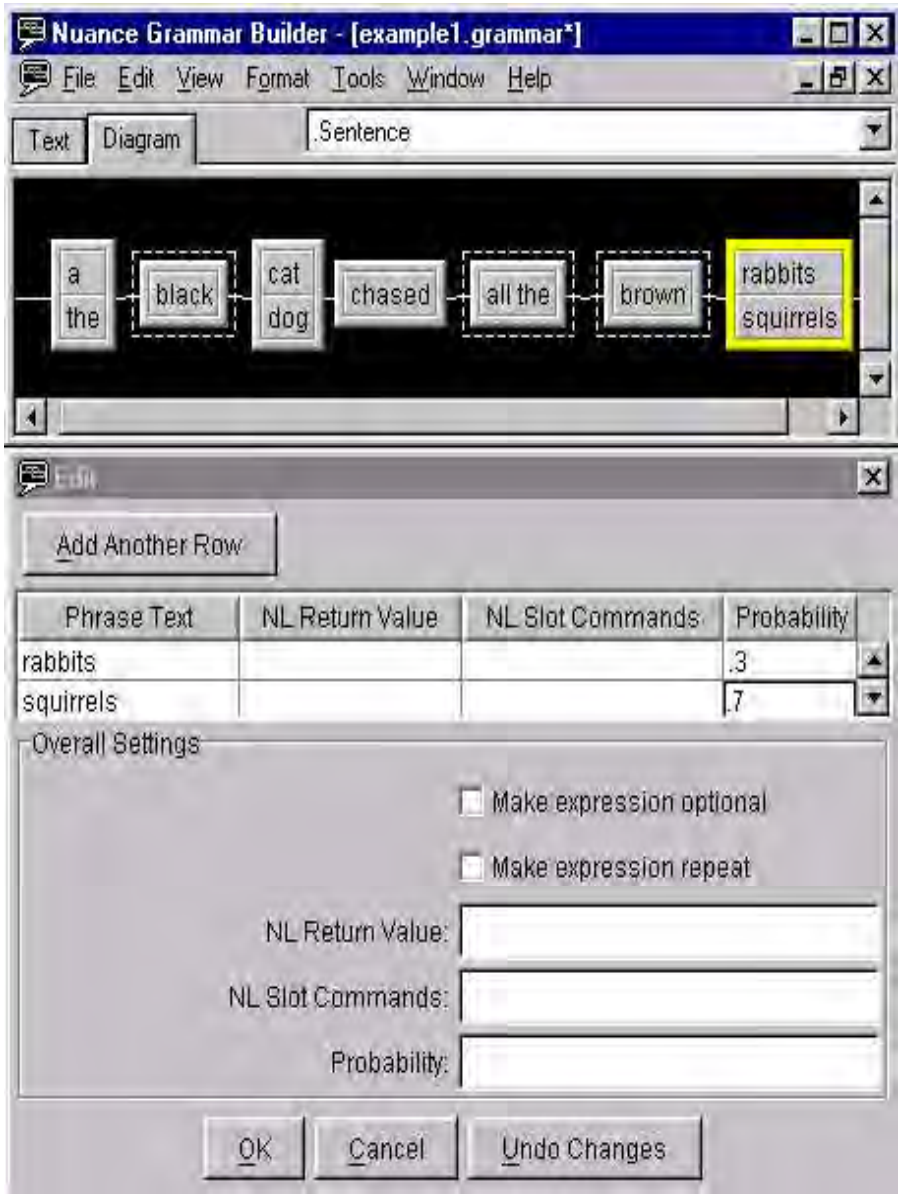
Assigning probabilities within NGB

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

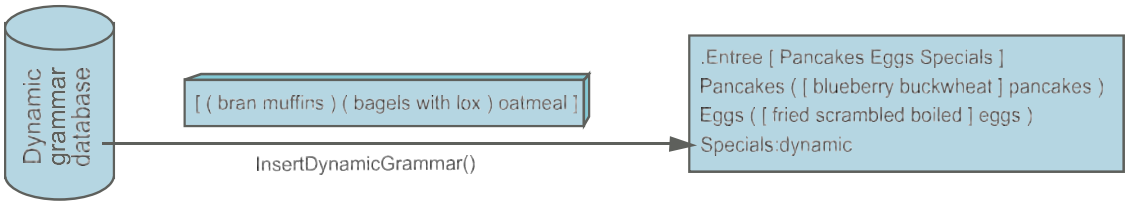
To add a dynamic grammar as a component in a top-level grammar, use the following GSL syntax:

```
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*.

Chapter 3

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 features 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)             aj 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.