Nuance VoCon Embedded Development System Development Formalisms

NUANCE Communications International BVBA
Guldensporenpark 32
B-9820 Merelbeke, Belgium
Phone: +32 9 239 8000 Fax: +32 9 2398001

Document: Nuance VoCon Embedded Development System Development Formalisms

V3.2, June 2009

Contents

INTRODUCTION

Nuance® VoCon® Embedded Development System (EDS) provides two development formalisms. They are not related to the actual application coding but they are important when you are designing your speech application. The first formalism is L&H+ and the second is BNF+.

The L&H+ phonetic transcription system is the vehicle for all the standard phonetic input and output to and from the recognizer. The VoCon automatic speech recognition (ASR) engine performs phonetic recognition, meaning it actually recognizes strings of phonemes.

The Nuance BNF+ grammar formats are the formal syntax that you have to use when writing VoCon ASR native source grammars.

If you are not familiar with VoCon ASR programming or with VoCon EDS and its features, please read the *VoCon EDS Developer's Guide*.

This manual contains the following sections:

- *The L&H+ phonetic transcription system* describes phonetic input in general. The phoneme tables for the different languages supported by the VoCon ASR engine are delivered in the add-on packages to VoCon EDS.

- *The Nuance BNF+ grammar formats* provides a comprehensive description of the grammar formalism. The concepts are introduced gradually, and many examples are provided along the way.

THE L&H+ PHONETIC TRANSCRIPTION SYSTEM

Introduction

Phonetic transcriptions are an essential tool in the description of human speech. The aim of a phonetic transcription is to provide a method of representing pronunciation in a written form in an unambiguous manner.

The most widely used phonetic alphabet is the one established by the International Phonetic Association (IPA). This alphabet consists mainly of the letters of the Roman alphabet together with a few letters from the Greek alphabet. A few other symbols are especially designed for IPA. The general principle is that *a distinctive sound is represented by one and only one symbol*. The IPA system also provides a set of *diacritics* to be added to the basic symbols. The purpose of these diacritics is to allow a finer description of the sounds if required.

The set of IPA symbols is not available on standard keyboards, and it is not supported in common computer character sets such as ASCII. To overcome this problem, Nuance has designed an encoding system that allows for the representation of the complete IPA character set using a limited range of ASCII symbols. This encoding system is called L&H+.

L&H+ covers all IPA symbols in the 1993 version, but it also adds a few symbols to represent phonetic elements not included in the IPA system. L&H+ uses members of the ASCII set in the range 33-126 to construct a phonetic symbol.

In L&H+, the percentage symbol “%” separates the basic symbol from the diacritics. For example, the IPA symbol /E)/ is /E%~/ iṇ L&H+

The language add-one packs to the EDS contain tables that provide lists of phonemes for several languages with their corresponding transcriptions in the L&H+ and IPA systems. Written and transcribed examples per phoneme are included.

Multiple pronunciations

The orthographic representation of a word can sometimes have completely different pronunciations. For instance, the word ‘*the*’ can be pronounced as /#'D$#/ or /#'Di#/ (depending on the context). Another example is the word '*different*', which can be read as /#'dI.f$.R+$nt#/ but also as /#'dI.fR+$nt#/. The difference between the two transcriptions is the deletion of /'$'/.

Note:

- In this section, phonetic transcriptions are written between forward slashes (‘/’). These are *not* part of the phonetic transcription; they are used here only for clarity.

In the pronunciation dictionaries (also known as exception dictionaries) and BNF+ grammars, multiple pronunciations can be specified in two ways.

The recommended way is to specify an L&H+ transcription for each pronunciation variant. For example, with a !pronounce statement in a BNF+ grammar (see below for more details), the two pronunciations for ‘the’ are specified as follows:
!pronounce the “#'D$#” | “#'Di#”;

It is also possible to specify multiple pronunciations within one L&H+ transcription. However, Nuance does *not* recommend using this technique because it is more complex and error prone. It is documented here and supported for backwards compatibility. Within one phonetic transcription, individual alternative pronunciations can be separated by the pipe symbol (‘**|**’) and enclosed in parentheses. The action of deletion is represented by the deletion symbol (**=0**).

Applying this formalism to the words *the* and *different* yields these transcriptions:

‘*the’*	/#'D ($ | i)#/
‘different’	/#'dI.f ($ | =0).R+$nt#/

This formalism should be used with care. The combination of alternative pronunciations can lead to the generation of invalid strings of phonemes or sounds. For example, the word ‘*record*’ can be transcribed either as /#R+I.'kOR+d#/ (the verb *to record*) or as /#'R+E.kOR+d#/ (the noun *record*). Combining these two representations into /#(' | =0) R+ (I | E).(' | = 0)kOR+d# / can generate 8 possible combinations.

One correct way to reflect the alternatives in a phonetic transcription based on our notation system is /#(R+I. ' | 'R+E.) kOR+d#/. If you want to specify more than two alternatives, you should group them two by two. This way, a phonetic transcription for the number *‘0’* could be the following:
/((# 'o&U# |# 'zi:.R+o&U#) |# 'zI.R+o&U#)/
This transcription combines three possible pronunciations for the number *‘0’*.

Inside pronunciation dictionaries, the recommended way is not to use the pipe symbol (‘**|**’) syntax at all. It is recommended to have multiple entries in the dictionary with the same key but different values. The dictionaries then return multiple transcriptions for this particular word.

The symbol # surrounding the phonetic transcriptions

In the examples in the preceding section, the phonetic transcription of each pronunciation variant starts and ends with the utterance boundary marker #. We recommend that you apply this convention systematically for all the pronunciations that are used by the VoCon ASR engine. The acoustic models contain special units for frequently used short words, such as digits and letters, to improve their recognition accuracy. The default phonetic transcriptions, including utterance boundary markers, are instrumental for enabling the engine to select these special units. When you are overruling default phonetic transcriptions for digits or letters, we recommend adding the default transcription, including the utterance boundary markers, as one of the alternative pronunciations. If you do not take this precaution, only phonetic models will be used, which may lead to poorer recognition accuracy for these words.

For example, when you want to add the alternative pronunciation /'na&I.n$R+/ for the digit "9" (for the pronunciation "niner"), use the expression (#'na&In#|#'na&I.n$R#). If you omit the utterance boundary markers and write ('na&In|'na&I.n$R) instead, recognition accuracy is likely to be poorer because the engine will not detect 'na&In as a special unit.

Phoneme tables

Note that phonemes in one language may not be available in another one. Phonemes that are not available in a given language cannot be used in a phonetic transcription for that language. The phoneme /i/, for example, exists in American English but not in German. Thus, a phonetic transcription generated by the American English grapheme-to-phoneme expert (G2P) system cannot be used as such in a German grammar with, for example, the !pronounce directive in a BNF+ source grammar. However, you can use a German grammar for the recognition of English words by replacing each of the unsupported English phonemes with its closest German equivalent.

THE NUANCE BNF+ GRAMMAR FORMATS

Introducing the Nuance BNF+ grammar formats

This section describes the Nuance BNF+ grammar format that is used to write continuous speech recognition grammars. This secton also describes how the functionality expressed by the Nuance BNF+ grammar formats fits under the VoCon API.

Grammar-related concepts

No matter how diverse a speech recognition application is, the phrases understood by a continuous speech system are, as a rule, defined by something called a context. A context is a series of rules that determines which utterances are to be recognized on a speech recognition engine. It can represent anything from a single word to a highly complex structure for a dictation recognizer.

BNF+ grammar format buffers are used to build grammars. A context can be made up of one or more grammars that are merged together. A diagram of the organization of the three primary speech recognition objects (recognizer, context, and grammar) is shown in Figure 1.

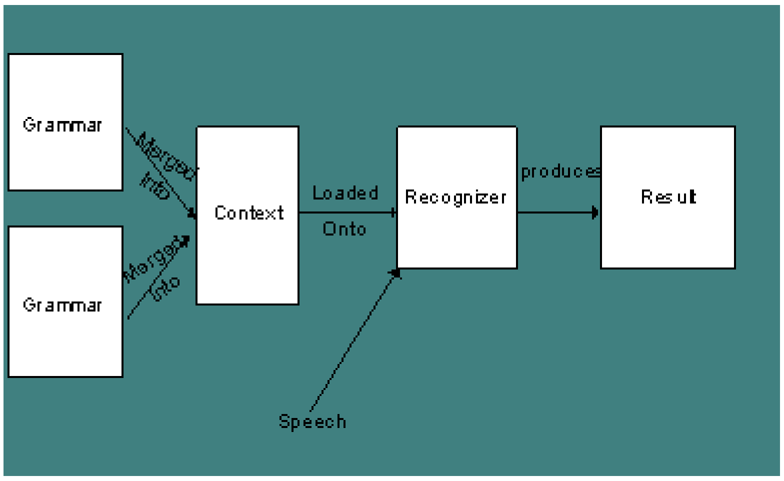

Figure 1: Relationship between speech recognition objects

In some speech recognition systems, this distinction between grammar and context is abstracted and either the grammar or context is not mentioned.

This chapter explains how the use of the Nuance BNF+ grammar format affects the contexts built with these grammars, the engine on which these contexts are loaded, and the results that are produced by this engine.

The Nuance BNF+ engine mode grammar format

Grammar essentials

The core of the BNF+ grammar format is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. An example of a grammar rule might be as follows:

<date> : the <day> of <month> <year> ;

Here <date>, <day>, <month>, and <year> each represent rules; <day>, <month>, and <year> must be defined elsewhere. With the proper definitions, the input phrase "the fourth of July 1776" will match the above rule.

In a grammar, any symbol that represents another set of symbols is called a non-terminal symbol, or non-terminal. In this rule definition, the symbols <date>, <day>, <month>, and <year> are all non-terminals. Any symbol that is complete in and of itself (that is, it needs no further specification) is called a terminal symbol, or terminal. In this rule definition, the words "the" and "of" are terminals. A terminal symbol is sometimes referred to as a word (even though it may in fact be a quoted string of several words, like "New York City").

Notation

The BNF+ grammar formats use a C-like syntax:

- Grammars are composed of statements terminated by a semicolon (‘;’).
- C++-style commenting is supported (both /* */ and //).
- Parentheses (‘( )’) can be used to group expressions together.

Statements, rules, and symbols

Imagine a snack bar in the not-so-distant future. Naturally, the computer needs to recognize the customers’ orders. The grammar below could be used in such a situation:

```
!grammar Drinks;

 !language "American English";

 !start <Speech>;

 <Speech>: lemonade | milkshake | orange juice;
```

This grammar should not be too hard to understand, and it clearly allows users to order lemonade, a milkshake, or orange juice. Nevertheless, further explanation is warranted.

The first three lines of the grammar start with an exclamation mark (‘!’) followed by a word. These words are directives—that is, words that have a special meaning to the grammar compiler.

This grammar also contains non-terminals, and, as previously described, they represent rules. Non-terminals can be any combination of characters from the character set, excluding the "greater than" symbol (">") and the "less than" symbol ("<"), enclosed by angle brackets (‘< >’). There are three reserved non-terminals: <VOID>, <NULL>, and <...>. The function of each of these non-terminals is explained later. When non-terminals appear at the beginning of a statement (in front of a colon ‘:’), they define a rule; when they appear in the middle of a statement (after a colon) they refer to that rule. In the sample grammar for the drink orders, only one non-terminal can be recognized: <Speech>. A terminal can be a string enclosed by double quotes (for example, "The Recognition Bar") or any sequence of (non-white space) characters except those listed in Table 2 (such as the word "Drinks" in our example).

Table 2: Restricted symbols in non-quoted terminals

‘ ‘	/	\	“	>	<	\0x0		\t	[	]	{
}	(	)	;	:	|	*	!	+	,	\r	

Let us now take a closer look at the first statement. This kind of statement is called a grammar statement. It contains the directive !grammar, followed by the grammar name, which has the same format as a valid terminal (so either a single identifier [‘Drinks’, in this example]) or any string within double quotes). It ends with a semicolon (;).

Grammar statements are required in the grammar format version 1.1. The combination of a grammar name and a slot name is used to uniquely identify the slot.

The second statement is called a language statement. The directive !language is followed by the language name, which has the same format as a terminal ("American English" in this example). It ends with a semicolon. The language statement specifies the primary language contained in the document. The language statement is optional.

The third statement is a start statement. Start statements begin with the directive !start, followed by one or more non-terminals and a semicolon. Start statements provide entry points into the grammar. Thus, they define what can be recognized. In this case, it is the rule identified by <Speech>.

This leaves us with the final part of this grammar: the definition of the rule <Speech>. This is a rule statement. A rule statement is made up of a non-terminal (sometimes called the "rule name") followed by a colon, which is followed by the definition of the rule and a semicolon. In this example, the right-hand side consists of three phrases, separated by the "or" operator

(‘|’). It indicates that the non-terminal <Speech> is defined as "lemonade" or "milkshake" or "orange juice." Each of these phrases is an alternative of the rule with the name <Speech>.

Since the start statement defines as in-context[1], by <Speech>, the conclusion is that only these three drinks can be recognized.

In the example above, the right-hand side of the rule <Speech> contains only terminals. In general, the right-hand side of a rule can also contain non-terminals or a mixture of terminals and non-terminals. The following grammar shows such a situation:

```
#BNF+EMV1.1;
/*
This grammar allows the user to order a hamburger and fries in addition
to the drinks. */
!grammar Order;

!language "American English";

!start <Speech>;

<Speech>:<Drink> | <Food>;

<Drinks>: lemonade | milkshake | orange juice;

<Food> : hamburger | "French fries";
```

Notes:

- All non-terminals in a grammar that are not specifically declared as imported (see *Importing rules: The !import and !export statements*) must be defined in that grammar.
- Within a grammar, a certain rule can be defined only once. The scope of a rule is local to the grammar where it is defined (unless it is exported; see *Importing rules: The !import and !export statements*). Therefore, the same rule name can be re-used in several grammars that are merged into one context, without introducing conflicts.

[1] Those utterances which can be recognized when this grammar is merged into a context and activated on an engine

Repeated recognition: The !repeat directive, the unary operators (+, *) and recursion

It is not unlikely that a customer would like to order multiple drinks. This can be done as follows:

```
#BNF+EMV1.1;
!grammar "3 Drinks";

!language "American English";

!start <Speech>;

<Speech>: <Drink><Drink><Drink>;

<Drink>: lemonade | milkshake | orange juice;
```

Although this works fine, there is a major drawback: the above grammar forces customers to always order three drinks. The following grammar overcomes this problem:

```
#BNF+EMV1.1;
!grammar "1 to 3 Drinks";

!language "American English";

!start <Speech>;

<Speech>: <Drink> | <Drink> <Drink> | <Drink><Drink><Drink>;

<Drink>: lemonade | milkshake | orange juice;
```

This is a good solution, but it again gives rise to problems if a range of up to 10 orders is desired. Of course, all possible combinations can be written, but this would be a lot of work. A better way is demonstrated in the grammar below:

```
#BNF+EMV1.1;
!grammar "1 to 10 Drinks Revised";

!language "American English";

!start <Speech>;

<Speech>: !repeat(<Drink>,1,10);

<Drink>: lemonade | milkshake | orange juice;
```

A new directive has just been introduced: !repeat(arg1, arg2, arg3). The first argument (arg1) is the expression to repeat, arg2 is the minimum number of iterations, and arg3 is the maximum number. The above grammar therefore models a situation in which a customer can order up to ten drinks.

Notes:

- The third argument can also be a star (*). This symbol is used to specify an undefined (unlimited) number of repetitions. However, if the maximum number of possible iterations is known, it is better to specify this number rather than using the star. There is no use in trying to recognize n+1 iterations if the maximum number spoken will never exceed n.
- The maximum number of repetitions should be larger or equal to the minimum number of repetitions.
- 0 is a valid value for the minimum number of repetitions.
- The first argument can be any top-comp-expression (a formal definition is given below).
- There are two other ways of phrasing a repeated expression in specific circumstances:

  Any expression can be followed by a * or a +. Expression* is a shorthand for !repeat(expression,0, *). Expression+ is a shorthand for !repeat(expression, 1, *). Also, expression* and !repeat(expression,*) are equivalent, just as expression+ and !repeat(expression,+) are. The unary operators can be applied on any expression (a formal definition is given below).
  Recursion:

  Simple right-recursion (the use of the rule name at the end of the definition of that rule) can be used to specify comp-expression+ in rules. For example,

  <rule>: <digit><rule>;

  and

  <rule>: !repeat(<digit>,+);

  are equivalent statements.

  The recursion that is allowed in the BNF+ grammar format is very limited in scope. If the name of the rule is used in its definition in any other position than at the end of an alternative, the building of the grammar will fail.

  Note:

  - This failure will occur in the flattening of the rule. Flattening will happen at grammar creation for non-modifiable and non-imported rules, and at context creation for modifiable and imported rules. The following are recursive examples that will fail:

```
<rule>: <rule> <digit>;
<rule>: <digit><rule><digit>;
<rule>: <rule>!optional(<digit>);
<rule>: <rule>;
<rule1>: <rule2><digit>;
<rule2>: <rule1>;
```

An example using all the concepts developed so far

The following grammar is a bit more complex using all the structures discussed so far. It lets the users say the number of each drink they would like, instead of forcing them to repeat the drink itself.

```
#BNF+EMV1.1;
!grammar "A lot of Drinks";

<Speech>: !repeat(<Order>,1, 3) please;

<Order>: <Single Order> | <Multiple Order>;

<Single Order>: one <Drink>;

<Multiple Order>: <Count> <Drinks>;

<Count>: two | three | four |five;

<Drink>: lemonade | milkshake | orange juice;

<Drinks>: lemonades | milkshakes | orange juices;
```

Valid orders would include:

one lemonade one orange juice please
three milkshakes five lemonades one orange juice please

But also the more unlikely:

one orange juice one orange juice please
five milkshakes two milkshakes one milkshake please
four orange juices one milkshake one orange juice please

The following are not valid:

one orange juice one milkshake one lemonade
one lemonades please

Note:

- This grammar models orders that are very unlikely to be given. As long as there is a chance they would be ordered, they should be modeled; otherwise the engine would not be able to recognize them. In such cases, this grammar would not be a good grammar, but for now and as a demonstration, it serves well.

Optional recognition: The !optional directive

Some of the above grammars include "please," and others do not. Since not all people use this word, there is a problem. Should it be added at the expense of poorer recognition for people who do not use it, or should it be left out, thus penalizing people who do use it? The obvious answer to this question is to allow both. However, just as with repetitions, it is not necessary to write down all possible recognition scenarios. As shown in the example below, a shortcut may be used:

```
#BNF+EMV1.1;
!grammar "Possibly Polite Drinks";

<Order>: <Drink> !optional(please);

<Drink>: a lemonade |a milkshake | an orange juice;
```

The above grammar has the same result as the one below:

```
#BNF+EMV1.1;
!grammar "Possible Polite Drinks Revised";

<Order>: <Drink>|<Drink> please;

<Drink>: a lemonade |a milkshake | an orange juice;
```

It may seem useless to introduce a whole new concept to the grammar-specification format when such a trivial workaround is available. In complex grammars, however, it will be an asset to have this option.

Notes:

- "!optional(x)" has the same effect as "(x | <NULL>)".
- Any comp-expression can be made optional.
- Square braces are a convenient shorthand for the directive !optional. "!optional(x)" can also be written as "[x]".

Importing rules: The !import and !export statements

It was stated earlier that each non-terminal referenced in a grammar must be defined in that grammar, with one exception. That exception is for non-terminals brought into the grammar from another grammar via the !import directive. When a non-terminal is imported, the compiler understands that the definition of that particular non-terminal will be coming from another grammar. When a context is created from a grammar containing imported non-terminals, that context should also include the grammar(s) exporting that non-terminal.

Here is an example of how this feature can be used:

```
#BNF+EMV1.1;
!grammar "snack shop";

!export <Food>

<Food>: pizza | hamburger | hot dog;
```

```
#BNF+EMV1.1;
!grammar "bar";

!import <Food>

<Order>: <Food> | <Drink>;

<Drink>: a lemonade | a milkshake | an orange juice;
```

The first grammar models a restaurant that serves hamburgers, hot dogs and pizzas. In the second grammar, this model is reused to model a bar that can serve food and drinks. When the context is created, both grammars need to be supplied to the grammar compiler.

Notes:

- An exported non-terminal is not automatically a start non-terminal, so if one wants to use the first grammar as a stand-alone, <Food> needs to be declared as a start non-terminal.
- Importing is useful for splitting the grammar specification over several grammars. It allows for the writing of modular grammars. Frequently used grammars (for example, for recognizing phone numbers and dates) can be written in separate grammars.
- Rules must be defined in the grammar from which they are exported. Rules cannot be defined in a grammar to which they are imported.

Functional note: Imports are resolved during context compilation in the tool *contextcpl.pyc.* All grammars must be present at that moment.

Pronunciations

The pronunciation of words used in a context can be specified in several different ways in Nuance BNF+ grammars. By default, the words used in the grammar are passed to a *grapheme-to-phoneme* (G2P) expert system, which converts them to sequences of phonemes. Alternatively, a dictionary of phonetic pronunciation (sometimes called a lexicon) can be used. If you want to use a dictionary,, you must indicate thus to the grammar compiler. Words for which there are transcriptions in the dictionary will not be passed to the G2P expert system. Pronunciations can also be specified directly in the grammar via !pronounce statements or directives. Pronunciations specified by !pronounce statements or directives take precedence over pronunciations specified in dictionaries or supplied by the G2P expert system. The next two sections describe !pronounce statements and !pronounce directives, respectively.

To summarize, the precedence order for the source of word pronunciation is:

1. !pronounce directives in the grammar
2. !pronounce statement in the grammar
3. Transcription supplied in the dictionaries used when compiling the grammar
4. Transcription provided automatically by the G2P converter

Grammar-wide pronounces: the pronounce statement

Imagine the addition of coffee to the list of drinks. This is a bit of a problem, because in American English there are two different pronunciations for this word: (in L&H+ format "'kO.fi " and 'kA.fi "). Normally, there should not be a problem because the phonetic expert system, the G2P module, together with the standard dictionary will automatically generate the most commonly used transcriptions for a word (for instance, 0 will be transcribed as "zero" and "oh"). But for coffee (and for some other words) this is not the case. Only the most general transcription will be used. If the alternative or perhaps both pronunciations are desired, the !pronounce directive must be used. The following sample grammar provides a possible solution:

```
!pronounce coffee "#'kO.fi#"|"#'kA.fi#";

<Drink>: lemonade | milkshake | orange juice | coffee;
```

In the above grammar, the pronounce statement tells the engine what phonetic transcription(s) have to be used for the word coffee in that particular grammar. The !pronounce directive should always be followed by the terminal (a single word or a double quoted string) for which you wish to provide the transcription. The second argument, the transcription itself, must be a quoted string preceded by an optional header.

Notes:

- Use the or-operator ("|") to enumerate alternative transcriptions. Each transcription is enclosed in double quotes.
- A header that specifies the type of transcription can precede each transcription. The default transcription type is the L&H+ phonetic alphabet. This transcription type uses the header L&H. Thus,

  !pronounce coffee "#'kO.fi#";

  is exactly equivalent to

  !pronounce coffee L&H "#'kO.fi#";

- A second way of specifying a transcription is by reference to another terminal. This transcription type uses the PRONAS header, which is an abbreviation for "pronounce as." Thus,

  !pronounce hi PRONAS "hello";

  means that hi must be pronounced as hello, or, in other words, the phonetic transcription(s) for "hello" will be the transcription(s) of "hi." This functionality can also be used to associate a terminal with its "userword" transcription. Userword transcriptions are transcriptions that are automatically created by the recognition engine in the course of userword training. Such a userword transcription can be added to a dictionary, together with its dictionary entry (this is usually the word for which a userword has been trained).

  With the help of the PRONAS header, these transcriptions can be associated with the terminals in the grammar. For example, suppose two userword transcriptions have been trained for the word "Jospin": one by an American (stored with the dictionary entry "Jospin_Am"), and another one by a French speaker (stored with the dictionary entry "Jospin_Fr"). In that case, the following pronounce statement specifies that the French pronunciation must be used throughout the grammar:

  !pronounce Jospin PRONAS "Jospin_Fr";

  The advantage of this approach is that the final association of (userword) transcription(s) with the actual terminal is not made in the dictionary, but, rather, in the grammar and can therefore be grammar specific.
- When a pronunciation for a certain terminal is looked up in a dictionary and that dictionary contains a userword transcription for that terminal, the transcription will be used only if its language and language version correspond with the language (version) used to open the grammar that contains the word.

Word-specific pronounces: The pronounce directive in rule definitions

The !pronounce directive can also be used within the rules. In this way, different instances of the same terminal within a grammar can have different transcriptions. For example:

```
!pronounce read "#'R+id#";

!pronounce record "#'R+E.k$R+d#";

<difPron>: I have read !pronounce("#'R+Ed#") a book
| We record !pronounce(L&H "#R+I.'kOR+d#") a record;
```

Notes:

- The !pronounce directive follows the terminal for which it specifies the pronunciation. Alternative transcription types and alternative pronunciations can be specified in the same way as in the pronounce statement.
- The pronunciations that are specified with a !pronounce directive on a specific instance of a word supersede the pronunciations that are specified for that word with a pronounce statement.

Character set and G2P

Grammars must be written in the UTF-16 or UTF-8 encoding of Unicode. Both encodings are accepted by the tool *contextcpl.pyc.* When passing a grammar to the VoCon API, it must be encoded in UTF-8. As mentioned before, the terminals in the grammar and the rule names can use any combination of characters from the character set except the reserved characters. However, there are some additional limitations on the character set used for terminals when the G2P expert system is called to generate pronunciations. Since the G2P expert system is designed to handle grapheme-to-phoneme conversion in a given language, it requires input sequences of characters that are acceptable words for that language in order to produce meaningful results. For instance, a French G2P module will not interpret Cyrillic characters correctly. If dictionaries or pronounce statements are used, the character set does not have these additional limitations.

Note on Japanese:

- For the Japanese language, two different types of characters are in use: *kanji* and *kana*. The G2P conversion module delivered in VoCon EDS supports only kana characters. More precisely, the general purpose RealSpeak G2P supports JEIDA-annotated kana, while the domain specific data-driven G2P supports Kana without annotations and Kanji. Please contact Nuance if you need additional information on the use of kana in VoCon EDS.

Word IDs: The !id directive

The !id directive allows the user to specify a Word ID for a terminal. The Word ID must be an integer. This feature can be used to make the grammar language independent or to process similar words in a uniform matter. The following example illustrates this:

begin !id(1) | start !id(1) | initiate !id(1)

As the same Word ID is specified for each of these words, the application doesn’t have to map them to the same meaning, because this is already specified in the grammar. Word IDs never appear in the recognized string.

Functional note:

- Word IDs can be found only by calling the lh_NBestResultFetchHypothesis function. See the *VoCon 3200 API Reference* for more information.

Spelling

The recognition engine allows you to spell words using the spelling feature. However, treating every letter as a separate word would yield poor recognition results because it is sometimes hard to recognize differences between individual letters—“m” and “n” for example. Here is an example of a spelling grammar:

```
!start <spell-syntax>; <spell-syntax>:!repeat(<letters>, 1,*);

<letters>: l | e |m |o |n |a|d|i|k|s|h|r|g|j|u|c|f;
```

Note that there is no mention of the words that are to be spelled in this grammar. Those words must be passed to the spelling post-processor. (See *application note on spelling post-processor* for more information on the spelling post processor.)

The tool *contextcpl.pyc* can create contexts of type SPELLING that configure the recognizer such that a special second pass is used, yielding a result type that can directly be passed to the spelling post-processor.

Allowing rules to be dynamically modified: The !slot statement

Sometimes the entire specification of the grammar is not known until runtime. In this event, the grammar needs to be modifiable, or, in other words, some of its rules need to be modifiable, and we must allow for the addition of terminals. Rules can be modified only if the !slot statement is used. An example of the use of the !slot statement follows:

```
#BNF+EMV1.1;
!grammar "Modifiable name dialing grammar"

!slot <names>;

!start <phone dialer>;

<phone dialer>: !optional(Please) call <names>;

<names>: <VOID>; /* See later: Special Rules: <VOID> */
```

Before the addition of names to the grammar, this grammar can never return a result. However, because the !slot statement was used, names can be added to the definition of the <names> non-terminal at runtime. The modification of the compiled context happens with the API call lh_BnfCtxSlotAddTermSeq.

Functional note:

- Rules should be defined as slots only if they need to be modified at runtime. When grammar-level optimization is used, this grammar optimization will not occur for the modifiable rules.

Allowing rules to be dynamically activated: The !activatable statement

There may be times when you want to recognize certain rules—for example, when you use a grammar with a wake-up word. When using a wake-up word, you can deactivate the entire grammar except for the wake-up word. After the wake-up word is recognized, the rest of the grammar is activated and the wake-up word is deactivated. The !activatable directive is used to specify which rules can be activated and deactivated.

The following is an example of a grammar using the activatable statement:

```
#BNF+EMV1.1;
!grammar "activatable example";

!start <top-rule>;

!activatable <wake-up><rest-of-grammar>;

<top-rule> : <wake-up>| <rest-of-grammar>;

<rest-of-grammar>: send e-mail | start web browser | [go to] sleep;

<wake-up>: computer;
```

Presumably, in this grammar, the user would like to activate the rule <wake-up> and deactivate the rule <rest-of-grammar> until the terminal "Computer" is recognized. Then the user will want to activate the rule <rest-of-grammar> and deactivate the rule <wake-up>. As soon as “[go to] sleep” is recognized, the reverse operation can take place, deactivating the rule <rest-of-grammar> and activating the rule <wake-up>. The API functions to activate/de-activate rules in the compiled context are lh_BnfCtxRuleActivate and lh_BnfCtxRuleDeactivate.

Functional notes:

- Start-rules are by definition activatable.
- The default state of a rule is "active". When a context is created in the VoCon API, all rules are active until they are de-activated.
- An imported rule can only be activated or deactivated in the grammar where it is defined and exported. Activating or deactivating an exported rule will cause that rule to be activated or deactivated in all grammars where it is imported.

Functional note: dynamic activation and the value of the Accuracy parameter

When parts of a context are deactivated, the Accuracy parameter (which controls the pruning of the search space) can be set lower than if the entire context were active. You can set a new optimal value for the Accuracy parameter (taking into account only the parts of the context that are active) by calling *lh_ConfigUpdateComputedParam* on the compiled context object with the parameter LH_CTX_PARAM_ACCURACY.

A second way to control when a new Accuracy parameter will be computed and applied is to call the function *lh_ConfigUpdateAllComputedParams* on the compiled context.

Special rules

The BNF+ grammar formats define three special rules, <...>, <NULL>, and <VOID>. These rules are universally defined and therefore don’t have to be imported. They cannot be redefined.

<...>

In the examples, several different ways to take an order have been modeled. But sometimes the exact phrasing is unknown. For example, you may know that a customer will order a drink from the menu, but not how the customer is going to phrase the order.

In such cases, it is only possible to check for specific words in a larger phrase. This is called keyword spotting. This method requires the garbage model (‘<...>’), which models any speech. Here is an example of its use:

```
!start <Order>

<Order>: <...><Drink><...>;

<Drink>: lemonade | milkshake | orange juice;
```

When this grammar is loaded, if an utterance occurs which contains the name of one of the drinks, it is recognized; this grammar will recognize only the first drink that is named in an utterance.

Notes:

- The use of the garbage model should be limited. in *VoCon 3200 API Reference*, see the description of the *garbage penalty* parameter for more information (*LH_REC_PARAM_GARBAGE*)
- Whatever speech or noise the garbage model absorbs will not appear in the recognition result. In the previous grammar example, only "lemonade," "milkshake," or "orange juice" can be the recognition result.
- The behavior of the garbage model in silence is determined by the language model used. In all current language models, silence matches the garbage model.

<NULL>

<NULL> is a special rule that is automatically matched. That is, it is matched without the user speaking any word. So the rule

<WithNull>: an <NULL> orange juice <NULL> please;

is exactly equivalent in recognition to the rule

<WithoutNull>: an orange juice please;

<VOID>

<VOID> is a special rule that doesn't match anything. That is, it can never be spoken. Inserting <VOID> into a sequence of grammar symbols automatically makes that sequence unspeakable. So the rule

<WithVoid>: an <VOID> orange juice please;

would never be matched in recognition.

Uses of <NULL> and <VOID>

In the BNF+ grammar format, empty rules and empty alternatives are disallowed. Instead, the grammar developer must use <NULL> or <VOID> in order to specify which behavior is wanted. So, again, empty definitions of rules and alternatives are illegal:

<rulename> : ; /* an illegal rule */

<another rule> : word | ; /* an illegal alternative, making the rule illegal. */

But definition of a rule or alternative as either <NULL> or <VOID> is legal.

<rulename2> : <NULL>; /* legal */

<rulename3> : <VOID>; /* legal */

<rulename5> : word | <NULL>; /* legal */

Another use of <NULL> and <VOID> is as a gating mechanism that can (for example) be used while developing grammars. By using one or more gate non-terminals in several places in the grammar, you can turn on or off several parts of that grammar by simply changing the right-hand side of a gate rule from <VOID> to <NULL>, or the other way round—for example:

```
#BNF+EMV1.1;
!grammar “Drinks”;
!start <Speech>;

<Speech>: !repeat(<Order>,1, 3) please;

<Order>: <Single Order> | <Multiple Order>;

<Single Order>: one <Drink>;

<Multiple Order>: <Count> <Drinks>;

<Count>: two | three | four | five;

<Drink>: lemonade |<gate> milkshake | orange juice;

<Drinks>: lemonades |<gate> milkshakes | orange juices;

/* change <VOID> into <NULL> in the rule below  in order to activate
milkshake(s) in the rules <Drink> and <Drinks> */
<gate>: <VOID>;
```

Note: This gating mechanism can also be used to emulate dynamic activations by means of dynamic modifications (by modifying the gate rule(s) dynamically). However, it is expected that most users will find the extensive set of dynamic activation capabilities more convenient. (See *Allowing rules to be dynamically activated: The !activatable statement*.)

Design tips

The more your grammar is adjusted to the situation, the better the recognition will be. The aim is to enable the engine to recognize all valid commands and only those. Making sure all valid commands are included is not too difficult. On the other hand, making sure that *only* those are included may require more effort.

Even if you have a good model of the speech that is likely to be produced, there are some other factors that may influence the quality of the recognition. Sometimes the differences in grammars are subtle, and grammars that seem to be the same may have different behavior. In the following sections, some of the thorny issues will be tackled.

Phrases

There are two things to consider in the selection of phrases for recognition. First, the longer the phrase is, the greater the amount of information that will be available and the more accurate the recognition result will be. Second, it is recommended that you use phrases that are as distinct as possible. The following example illustrates two phrases that are not very distinct:

I am going to the store vs. *I am going in the store*

As a rule of thumb, try to use longer and more distinct sentences.

Another problem is opposite words. Many of them sound very similar. Typically, the only difference between the two resides in prefixes like "un-" and "in-" (for instance, "popular" versus "unpopular"). It is good practice to avoid these kinds of opposites. If a good alternative to such opposites cannot be found, using them in longer sentences in which the “carrier” sentences themselves are sufficiently different may help differentiate between the two.

Quotation marks

Quotation marks (in the context of the paragraph this means double quotes) should be used with caution. The use of quotation marks around a sequence of words turns that sequence into a single grammar terminal, and as a result, the sequence is treated like a single word. This has a number of consequences. A sequence of words in double quotation marks will appear as if it is a single word in the recognition result, so there will be no confidence levels, nor segmentation information for the individual words within that sequence. Another consequence is that the optimizer of the grammar system cannot merge a word from one sequence in quotation marks with another occurrence of the same word in a different sequence in quotation marks or in isolation.

If quotation marks are used and if the phonetic transcription of the word sequence in quotation marks contains the symbol '_' in the position of the boundaries between the words, two speech unit transcriptions are generated from that phonetic transcription (speech unit transcriptions are lower-level, ASR engine-specific transcriptions that are derived from the phonetic transcriptions). One speech unit transcription will contain no pause model between the individual words, and the other will always contain the pause model between the words. In this way, the engine still has some capacity to handle pauses between words. However, this is less flexible than allowing optionally a pause between each of the words, as the engine would do if the sequence of words were not in quotation marks. In almost all cases, the G2P module will produce the '_' symbol on the word boundaries, so the behavior described above is the default behavior. However, the user can force the engine to either put or not put a pause model between the words that make up a string in quotation marks by specifying the appropriate phonetic transcription for the word sequence. Inside phonetic transcriptions, you can use the symbol '##' to force a pause model at that position. The symbol '_' (word boundary) will generate the two transcriptions as explained above. Avoiding these symbols will give you a transcription without pause model.

Phonetic transcriptions can be specified in the dictionary or with a pronounce directive or pronounce statement.

Interaction

A context-free grammar is not a dictation grammar: users cannot just say what they want. A grammar should be designed to handle specific syntaxes. The design must be consistent to enable users to become familiar with the context and anticipate expected responses. The use of well-chosen prompts can also help. The less confused users are about a possible response, the greater the chance that they will use a correct command and the higher the accuracy of the recognition will be.

It is also important to give feedback so that the users will know that their command has been correctly understood. Feedback should not be overdone, however, because this can slow down the dialog. Many of the best systems incorporate feedback into the dialog.

Limit the use of "special features"

Although special features may help you create the perfect grammar, their use often restricts the optimization of the grammars, resulting in longer response times, and decreased accuracy.

The garbage model is somewhat special. It is actually ”dangerous,” as it models any speech. If you know what is going to be said, even though you are not interested in it, listing this speech will provide better performance than using the garbage model. Even if you know only a limited part of the text that must be spoken, that text should be added with the garbage model. Generally speaking, the more speech you can identify, the better the results will be.

Similarly, avoid overusing repetitions. If you know the maximum number of repetitions, it is better to specify this number than to specify just *. Trying to have the engine recognize 8 digits in a 7-digit telephone number serves no purpose. Note that specifying the maximum number of repetitions adds more memory to the grammar; this is especially true if there is a large range of repetitions (for example, !repeat(<digit>, 1, 20)). If you are using a large range, it is recommended that you use an infinite maximum number because of memory considerations.

Optimization

If there are common parts to a grammar, it is recommended that you separate them from the specific parts, as in the following example:

What’s your name | What’s your address;

What’s your (name | address);

In this example, both lines will recognize the same two utterances, but they may have a different effect on the flattening of the grammar and the number of words sent to the recognition engine. The two options below may also have a different effect on flattening:

<digit><digit>|<digit><digit><digit>;

<digit><digit>!optional(<digit>);

Vocabulary management

As explained earlier, only those phrases that are meaningful to recognize at a given time should be active. Often, a context is a general representation of the speech that can be spoken during the life span of an application, and no context switching goes on during the application’s lifetime. This, however, does not necessarily mean that all of the words in the application’s context have a valid meaning all of the time. When possible, the application should limit the number of active words in the grammar at runtime. The fewer the number of words that are active at any point in the grammar, the lower the branching factor will be; thus, the engine will be able to recognize the words more easily.

Other elements that facilitate vocabulary management are *activatable* rules, or rules that can be activated or deactivated.

Processing Mandarin Chinese with the VoCon ASR engine

This section contains some specific information for processing Mandarin Chinese with the VoCon ASR engine. Although not contained in the VoCon EDS default installation package, a language add-on package is available for Mandarin Chinese.

Vocon ASR acoustic models for Mandarin Chinese have been trained on speech data from speakers from the following regions/accents: Beijing, Shanghai, Zhejiang, Anhui, and Jiangsu provinces. Special focus has been given to the Beijing and Shanghai regions, meaning that a relatively higher proportion of the speakers come from one of these two regions.

Writing in Chinese

In the Chinese language, words can be written either in Chinese characters (traditional or simplified) or in pinyin (with or without tones, but it’s strongly recommended that you add tone information). Pinyin is a romanization system for standard Mandarin and usually refers to Hanyu pinyin. The Republic of China on Taiwan is in the process of adopting a modified version of pinyin (currently Tongyong pinyin). In early October 2000, the Mandarin Commission of the Ministry of Education proposed using Tongyong pinyin as the national standard, but it was rejected. Tongyong pinyin is similar to Hanyu pinyin with a number of changes in the letters. To find out the differences, consult the following link: http://research.chtsai.org/papers/pinyin-xref.html In short, pinyin can be seen as a kind of phonetic transcription written in roman characters. Here are two examples:

Pinyin: zhong guo (Zhongguo) (Middle Kingdom = China)
Pinyin with tone: zhong1 guo2 (Zhōngguó)
Traditional Chinese characters (hànzì): 中國
Simplified Chinese characters (hànzì): 中国

Pinyin: long li (Long Li) (Pretty Dragon, person’s name)
Pinyin with tone: long2 li3 (Lóng Lǐ)
Traditional Chinese characters (hànzì): 龍麗
Simplified Chinese characters (hànzì): 龙丽

All Chinese people can write in pinyin, because it is actually the first writing alphabet that they learn at school. Since Hanyu pinyin is ISO accepted for modern Chinese, the VoCon ASR grapheme-to-phoneme module (G2P) expects this type of pinyin with some additional rules:

- **Use tones (1 to 5)**. The tones 1 to 4 are used as standard. For the neutral tone, tone 5 has to be used (not 0, as is sometimes used in literature). Never use tone 0 in the VoCon ASR engine.
- **Rhoticized pinyin** input should always be written with **r' after the tone information** (for example, feng1r', *not* feng1r and *not* fengr1)
- Use the **vowel u to represent the vowel ü** if the same syllable does not exist with the vowel u (xuan1, quan1, yun1, jun2). In the other cases, you should **use the letter v to represent ü** to make the distinction between u and ü:
  - nv1 != nu1
  - lv1 != lu1

For the VoCon ASR engine, not writing the tone information is equivalent to writing a neutral tone (that is, zhong and zhong5 are equivalent). It is important to write the tone information. Not marking the correct tones will lead to a relative increase in error rate that can vary from zero percent to several tens of percent, depending on test grammar and test utterances.

Chinese in BNF grammars

As mentioned above, the VoCon ASR automatic grapheme-to-phoneme (G2P) conversion system accepts pinyin tone as input. It is not possible to automatically convert traditional or simplified Chinese symbols using the automatic G2P module. This means that the BNF grammars can be written in Hanyu pinyin with tones. However, if hànzì characters are needed in the string result (for example, for display purposes), the application designer has the following possibilities:

- using !pronounce statements or directives in the BNF grammar (works with context from grammar and context from buffer functionality):

  !pronounce "中国" PRONAS "zhong1 guo2" / PinYin */*

  !pronounce "中国" "t&s+o55nK.kwO35" / LH+ */*

  By using one of these statements, the word "中国" can be used directly in the BNF and the string result will contain the hànzì symbols. In the first case, the Hanyu pinyin transcription is converted by the G2P; in the second case, the G2P is overruled by the phonetic transcription in the grammar.
- using a user dictionary to convert Chinese characters into LH+ (works with context from grammar and context from buffer functionality):

"中国" + user dictionary

This is similar to the !pronounce statement above. The dictionary can contain both Hanyu pinyin transcriptions and LH+ phonetic transcriptions.

DDG2P

The data-driven g2p module for Mandarin actually supports hanzi at the input. For run-time addition of words to slots using the DDG2P, hanzi characters can be used.

Processing Cantonese with the VoCon ASR engine

For processing Cantonese with the VoCon ASR grammar system, the same comments apply as in the section about Mandarin Chinese. In order to support Cantonese in the BNF grammars, you have exactly the same options. What is different with respect to Mandarin Chinese is the writing system. The Nuance RealSpeak™ G2P module shipped with VoCon EDS supports jyutping as a writing format.

Jyutping

In Cantonese there are two writing formats: simplified and traditional. In Hong Kong, people use the traditional system, while in Guangzhou, the simplified one is used. The two systems have different sets of characters, but the pronunciation is the same.

There are several romanization systems used for teaching the Cantonese language and other purposes. The most widely used is the Yale romanization system. The newest system is the one proposed by the Linguistic Society of Hong Kong. It is called Jyutping and differs from the Yale romanization system as follows:

- Jyutping marks six distinct tones with the numbers 1 to 6 following the syllable; the Yale 0 and 1 tones merge into the tone marked by 1 in Jyutping.
- In the initials, Jyutping uses "c" for the Yale: "ch"; "z" for the Yale "j"; and "j" for the Yale "y".
- In the finals, Jyutping uses "aa" for the Yale "a"; "oe" for the Yale "eu"; "eoi" for the Yale "eui"; "eon" for the Yale "eun"; "oeng" for the Yale "eung"; "eot" for the Yale "eut"; and "oek" for the Yale "euk". Jyutping also includes distinct "eu", "em", and "ep" finals.

The initial, finals, and tones defined in the Jyutping system can be found at http://en.wikipedia.org/wiki/Jyutping.

The Cantonese language has a somewhat complicated tone system. In Guangzhou, seven different tones are distinguished and there is a tendency to merge the high-level and high-falling tones into one. The Hong Kong dialect has nine distinct tones: high-level (1), high-rising (2), mid-level (3), low-falling (4), low-rising (5), low-level (6), and three more entering tones—high, mid, and low. The entering tones are the same as 1, 3, and 6, but appear in closed syllables—that is, in syllables that have “p”, “t”, and “k” as syllable coda. Thus, we can talk about six distinct tonemes and the “entering” tones as tone variants appearing in specific environments (so called allotones). The RealSpeak G2P system also accepts six different tones as defined in the Jyutping romanization system, starting from tone 1.

BNF+ ENGINE MODE

Formal specification

Basic types in engine mode

non-terminal
any string of characters between < and >
quoted-string
any string of characters between a pair of quotes.
word
any string of characters except those in the following table

Table 6: Restricted symbols in non-quoted terminals

' '	/	\	"	>	<	\0x0		\t	[	]	{
}	(	)	;	:	|	*	!	+	,	\r	

terminal
Either a word *or a* quoted-string
integer
A number that can be negative, but cannot be fractional. Positive integers cannot be signed (cannot have a + prepended).

Precedence levels and associativity for operators

The following preceding levels and associativity rules apply:

Table 7: Precedence Order in Nuance BNF+

Precedence	**Operator**	**Associativity**
Highest	Directives	
	Unary operators	

	Parentheses, brackets	
	Sequence	left, binary
Lowest	Alternative	left, binary

Directives have the highest precedence level. The binary, left associative operator '*concatenation*' (which does not have a symbol, but is used to create a sequence) has a higher precedence level than the binary, left associative operator *alternative* "|". The implicit precedence levels can be overruled by the use of parentheses.

Examples:
New York | *Amsterdam*
New (*York* | *Amsterdam)*

The implicit precedence levels make the first example an alternative list of "New York" and "Amsterdam" where the second example is an alternative list of "New York" and "New Amsterdam".

Nuance BNF+ engine mode grammar format formal specification

Notes:

- C/C++-like comments "/* */" and "//" can be used in BNF grammar files. Both comments and whitespace are allowed anywhere between tokens.
- The quotation marks in the following specification are used to denote literal strings. Quotation marks are only valid characters in this grammar format in comments and in the quoted-string token.
- The "grammar" non-terminal described directly below describes the complete specification grammar:

HEADER *sequence-of-statements*

HEADER(in ASCII):
"#BNF+EMV1.1;"
<empty> ";"

sequence-of-statements :
<empty>
statement
sequence-of-statements

statement :
language-statement ";"
export-statement ";"
import-statement ";"
pronounce-statement ";"

start-statement ";"
rule-statement ";"
activatable-statement ";"
slot-statement ";"
grammar-statement ";"

grammar-statement :
"!grammar" *terminal*

language-statement :
"!language" *terminal*

export-statement :
"!export" opt-par-non-terminal-seq

import-statement:
"!import" opt-par-non-terminal-seq

start-statement:
"!start" opt-par-non-terminal-seq

slot-statement :
"!slot" opt-par-non-terminal-seq

activatable-statement :
"!activatable" opt-par-non-terminal-seq

opt-par-non-terminal-seq:
non-terminal
par-non-terminal-seq
opt-par-non-terminal-seq non-terminal
opt-par-non-terminal-seq par-non-terminal-seq

par-non-terminal-seq:
"(" non-terminal-seq ")"
"(" par-non-terminal-seq ")"

non-terminal-seq :
non-terminal
non-terminal-seq non-terminal

pronounce-statement:
"!pronounce" *terminal* pronounce-alt

pronounce-alt:
header quoted-string pronounce-alt "|"header quoted-string
header quoted-string

header:
"L&H"
"PRONAS"
<empty>

rule-statement:
non-terminal ":" top-comp-expression

top-comp-expression:
comp-expression "|" top-comp-expression
comp-expression

comp-expression :
expression
comp-expression expression

list-of-terminals :
terminal
terminal "," list-of-terminals

expression :
terminal id-operator pronounce-operator
non-terminal
unary-expression
parent-expression
operator-expression

id-operator :
"!id" "(" *integer* ")"
<empty>

!pronounce-operator :
"!pronounce" "(" pronounce-alt ")"
<empty>

paren-expression :
"(" top-comp-expression")"

operator-expression :
repeat-operator
optional-operator

unary-expression :
expression "+"
expression "*"

repeat_operator :
"!repeat" "(" top-comp-expression "," *integer* "," *integer* ")"
"!repeat" "(" top-comp-expression "," *integer* "," "*" ")"

"!repeat" "(" top-comp-expression "," "*" ")"
"!repeat" "(" top-comp-expression "," "+" ")"

optional-operator:
"!optional" "(" top-comp-expression ")"
"["top-comp-expression "]"

Statements

Grammar header

Format
#BNF+EM V1.1;
Description
The grammar header specifies the type of grammar (in this case BNF+ engine mode) and the version.
Example
#BNF+EM V1.1;
Comments
- For the BNF+ engine mode version1.1, the grammar header is required, but recommended. It must be specified on the first line of the grammar file. No spaces may precede it.
- The character encoding of the header is the same as that of the entire grammar, but all characters used in the header must be from the ASCII subset of the header being used.

!language statement

Format
!language *terminal* **;**
Description
The !language statement specifies the language of the grammar.
Example
!language "American English";
Comments
- This is an optional statement: the language will be specified anyway as a parameter to the grammar compiler.
- Any string can be used in this field. It is documentation for the grammar writer only.

!pronounce statement

Format
!pronounce *terminal* [*word*] quoted-string (| [*word*]*quoted_string*)***;**

Description
With the !pronounce statement you can override the default phonetic transcriptions for certain *terminals*. When such statements are given for a certain *terminal*, the given transcriptions are used instead of the default phonetic transcriptions for that *terminal*. The given transcriptions are used for all occurrences of that *terminal* in the grammar (unless they are over-ruled by a pronounce directive on a particular occurrence of the *terminal*, see further). The phonetic alphabets for specific languages can be found in the appendix. The *word* preceding the *quoted_string* in the format is the

header to the pronunciation, and specifies the type of pronunciation that follows. In the current specification, two headers can be used: L&H (which is the default) and PRONAS. The PRONAS header (which is an abbreviation of "pronounce as") is used to specify that the phonetic transcription(s) of a *terminal* are the same as those of the *terminal* enclosed in the double quotes.

Example
!pronounce Dr "# ' d A k . t $ R+ #" | "# ' d A k #";
!pronounce St L&H "# ' s t R+ I t #";
!pronounce Jospin PRONAS "Jospin_UW"

Comments

- If no header is specified, the phonemes must be part of the L&H+ phonetic alphabet, defined for the specific language. Multiple phonetic transcriptions can be specified in one pronounce statement. They have to be separated with a "|".
- Care must be taken for special symbols inside the transcriptions. Since they are double quoted, the same rules apply as for double quoted terminals. This implies that the character "\" is treated as an escape character and hence has to be escaped itself if it needs to be interpreted literally (for example, the LH+ symbol “h\” has to be written as “h\\”).

!export statement

Format
!export non-terminal+**;**

Description
The !export statement allows you to export frequently used *non-terminals* for use by other grammars. Other grammars can import the *non-terminals* by linking the grammars.

Example
!export <Number>;
!export <Integer>;

Comments

- The exported *non-terminals* must be defined in the grammar.

!grammar statement

Format
!grammar *terminal* **;**

Description
The !grammar statement specifies the name of the grammar.

Example
!grammar TestGrammar;

Comments
- This is a required statement in BNF+ engine mode version 1.1.

!slot statement

Format
!slot non-terminal+**;**

Description
Specifies which rules are modifiable and for which the right-hand side can be modified.

Example
!slot <rule1> <rule2>;

Comments
- The !slot directive should be used as sparingly as possible. In the event of grammar optimization, the modifiable rules will not be optimized See *VoCon 3200 API Reference* for a full description of the dynamic modification capabilities of its recognition engines.

Rule statement

Format
non-terminal **:** *comp-expression* **;**

Description
The non-terminal to the left of the colon is the name of the rule; comp-expression represents a regular expression containing *terminals* and *non-terminals*; the colon and the semicolon are Nuance BNF+ punctuation. A rule statement specifies a set of *terminal* strings to be matched. It contains *terminals* (which match words to be recognized), *non-terminals* (which match symbols defined by rules in the grammar), operators and directives (which specify repetitions, choices, and other features).

!start statement

Format
!start non-terminal+**;**

Description
The !start statement allows you to specify which *non-terminals* correspond to top-level rules. Only utterances that satisfy the syntactical specification of at least one top-level rule can be recognized.

Example
!start <Number>;

Comment

- The format allows you to put parentheses around some of the *non-terminals*.

!import statement

Format
!import non-terminal+**;**

Description
The !import statement allows you to import frequently used *non-terminals* from other grammars. Only those *non-terminals*, which have been exported from other grammars, can be imported.

Example
!import <Number>;
!import <Month>;

Comment

- The non-terminal symbols specified in the !import statement cannot be defined in the grammar itself.
- The format allows you to put parentheses around some of the *non-terminals*.

!activatable statement

Format
!activatable non-terminal+ **;**

Description
Specifies which rules are activatable at runtime using rule activation.

Example
!activatable <rule1> <rule2>;

Comment

- The functional interface has the option to specify at grammar object creation time that all rules are activatable. This option supersedes any possible !activatable declaration in the grammar. If each rule in the grammar is activatable (that is, each rule in the grammar can be turned on or off at runtime), the context that will be loaded on the recognition engine cannot be optimized as much as in the case none of the rules are activatable. This implies that the memory and CPU requirements of the recognition engine will be larger if full dynamic rule activation would be required. How much larger depends on the grammar(s). The directive !activatable allows the grammar developer to specify precisely which rules in the grammar need to be activatable. In this way, the context can still be optimized as much as possible while offering the dynamic rule activation capabilities that are needed. See *VoCon 3200 API Reference* for a full description of rule activation and deactivation.

Directives

!id directive

Format
!id(*integer***)**

Description
The !id directive specifies a Word ID (which must be an integer) for the preceding *terminal* . If the !id directive does not directly follow a terminal, a syntax error results.

Example
read !id(3)

!pronounce directive

Format
!pronounce(*[word]quoted-string* **(|***[word] quoted-string)** **)**

Description
The !pronounce directive specifies the phonetic transcription(s) of the preceding terminal. It overrules all other specification of pronunciations for that particular occurrence of the terminal in the grammar. The *word* in the pronounce statement is a header for the pronunciation, and is optional. The header specifies the transcription type. The default transcription type is the L&H+ phonetic alphabet. This is assumed if no header is specified. It can be manually specified by use of the L&H header. The quoted-string is the pronunciation itself. The second possible header is "PRONAS". This specifies that the quoted-string is not the transcription itself, but a *terminal* (which does not have to be elsewhere in the grammar) whose transcriptions will be used.
The !pronounce directive must directly follow either a *terminal* or an !id directive (which must directly follow a *terminal*). If it does not, a syntax error will result.

Example
read !pronounce("'R+Ed")

Comments

- If no header is specified, the phonemes must be part of the L&H+ phonetic alphabet, defined for the specific language. Multiple phonetic transcriptions can be specified in one pronounce statement. They have to be separated with a "|".
- Care must be taken for special symbols inside the transcriptions. Since they are double quoted, the same rules apply as for double quoted terminals. This implies that the character "\" is treated as an escape character and hence has to be escaped itself if it needs to be interpreted literally (for example, the LH+ symbol “h\” has to be written as “h\\”).

!repeat directive

Format
!repeat(*comp-expression***,** *integer* [**,** (***** | *integer*)]**)**
or
!repeat(*comp-expression***,** (***** |**+**)**)**

Description
The repeat defines an alternative of sequences growing in length by repeating the*(top-)comp-expression* given as first argument. A !repeat directive is itself an *expression*. In the first format, the minimal number of repetitions is the first integer, and the maximal number of repetitions is either the second *integer* given or infinite (if there is a second comma followed by *****). If the first integer is zero, then the !repeat is optional. If there is no second comma, then the minimum number of repetitions in this format is also the maximum number of repetitions. *expression** is a shorthand for !repeat(*expression*,*) and for !repeat(*expression*,0,*). *expression*+ is a shorthand for !repeat(*comp-expression*, +) and for !repeat(*expression,* 1, *).

Example
!repeat(<digit>, 9, 12)

Comments
- This stands for an alternative of 4 sequences, ranging from 9 to 12 <digit>'s.

!optional directive

Format
!optional(*comp-expression***)**

Description
The !optional directive makes a *comp-expression* optional in a certain rule. It is itself a *comp-expression*.

Example
the !optional(<adjective>) tree

Comments

- The non-terminal symbol <adjective> is optional in this *comp-expression*. There may be an adjective, but it is not necessary. Note that the notation:
  the (<adjective> | <NULL>) tree
  is entirely equivalent.

Other important concepts

Alternatives

Format
comp-expression **|** *comp-expression*

Description
Note that this description describes two alternatives. Alternatives are separated by "|". A list of alternatives is also itself a comp-expression. The following expressions can have alternatives, and **must** have at least one alternative:

- rule statement
- !optional directive
- bracketed expression
- !repeat directive.

Example
<Cities>: Boston | New (York | Amsterdam) | <Foreign Cities>

Comments

- In this example expression, the rule <Cities> has three alternatives: the word "Boston", the word "New" followed by the parenthetical expression ("York"|"Amsterdam") and the non-terminal <Foreign Cities>. The parenthetical expression in this example has two alternatives, "York" and "Amsterdam".

Bracketed expressions

Format
[*comp-expression***]**

Description
The *comp-expression* enclosed within square brackets is optional. An bracketed *comp-expression* is itself a *comp-expression*.

Parenthesized expressions

Format

(*comp-expression***)**

Description
A *comp-expression* enclosed within parentheses is itself a *comp-expression*.

Unary operator: +

Format
*comp-expression***+**

Description
Is the same as **!repeat(***comp-expression***, 1, *)**

Example
<number>: <digit>+;

Sequence

Format
comp-expression comp-expression

Description
A sequence of *comp-expression*s is itself a *comp-expression*. The value of this expression is the concatenation of the *comp-expression.* The white space between consecutive *comp-expressions*, *non-terminals* or words is the implicit concatenation operator.

Example
I want to go to <City>

Comments
- This example indicates that the words "I", "want", "to", "go", "to" and the non-terminal <city> are matched by this expression if they occur in this specific sequential order.

Unary operator: *

Format
*comp-expression******

Description
Is the same as **!repeat(***comp-expression***, 0, *)**

Example
<number>: <digit>;*

Pronunciation types

<u>The PRONAS Transcription Type</u>
The PRONAS transcription type allows the user to say that one word should be pronounced in the same manner as a different word. This different word's pronunciation will be checked in the user dictionaries associated with this grammar, and if not found there, in the expert system (G2P). The PRONAS transcription type is case-sensitive.

<u>L&H Transcription Type</u>
This is a phonetic transcription in the L&H+ phonetic alphabet. Transcription types are used in !pronounce directives and !pronounce statements. Space is not important in this alphabet, except in that certain symbols of this alphabet (such as R+ in the American English version) require multiple characters. These characters must be consecutive, without spacing.