Cerence ASR (Embedded) Grammar Formalisms

INTRODUCTION

The Cerence® Companion SDK provides several development formalisms. They are complementary to the actual application coding but are just as important when you are designing your speech application.

Before you read this document and if you are not familiar with Cerence ASR (Embedded) programming or with Cerence ASR (Embedded) and its features, please read the **`Cerence ASR (Embedded) Documentation for Companion SDK`**.

The formalisms described in this document is BNF+. The Cerence BNF+ grammar format is the specification language that you have to use when writing Cerence ASR (Embedded) native source grammars.

This manual provides a comprehensive description of the BNF+ grammar formalism. The concepts are introduced gradually, and many examples are provided along the way.

THE CERENCE GRAMMAR FORMAT

Introducing the Cerence grammar format

This section introduces the Cerence BNF+ grammar format that is used to write continuous speech recognition grammars. We also describe how the functionality expressed by the Cerence grammar format fits into the Cerence ASR (Embedded) API and configuration.

Contexts

An application with speech recognition will work best if it models it’s *expectations* of what the user is going to say. The speech recognition *context* is Cerence ASR (Embedded)’s abstraction which describes those expectations.

As explained in the developer’s guides, a context is a binary data file which is read by the Cerence ASR (Embedded) recognizer. It is specified when configuration a Cerence ASR (Embedded) application. For example, in the following JSON application configuration file, *modelm_small.fcf* is a context:

```
{
    "version": "VoConHigh 5.0",
    "application": [
        {
            "name": "APP_CNC",
            "input": [
                {
                    "type": "asr",
                    "asr": {
                        "search": [
                            {
                                "name": "CNC_CTX",
                                "type": "static",
                                "static": {
                                    "file_name": "modelm_small.f
                                },
                                "vocon_parameters": {
                                    "LH_SEARCH_PARAM_TSILENCE":
                                }
                            }
                        ]
                    }
                }
            ]
        }
    ]
}
```

The binary data of a context is either created off-line with one of the *context creation tools*, or created at runtime using a **Dynamic Content Consumer (DCC)**. Contexts that are created off-line can be much more complex and larger than those created at runtime.

Contexts describe what user can say because they contain *sentences*. A sentence is a sequence of *terminals*. A terminal is roughly the same as a word. The term terminal is used because a name like *New York* which contains spaces can also be considered 1 terminal whereas many people see it as 2 words because of the spaces. As we will see later we can also associate other information with terminals.

Cerence ASR (Embedded) has several approaches to the creation of binary contexts which can be divided in 2 categories.

The first category are the approaches which treat the context as a *list the* *set of possible sentences*. A particular sentence can be in the context or it is not in the context. If a sentence is not in the context then it can never be recognized! The set of sentences may be infinite, for example a sequence of digits of arbitrary length. Human language also contains potentially infinite constructs like the embedded clause. The sentence: *the car which is green has been owned by John and was constructed in 1979 and …* may go on forever.

Where the set of sentences in a context is potentially infinite, the set of terminals is always finite. The set of terminals in a context is often referred to as the *vocabulary* of a context.

If the user speaks a word (terminal) that is not in the context then we say the word is *out of vocabulary*, often abbreviated as OOV. Take for example a context which describes 1 sentence: *today it is warm outside*. This context contains 5 terminals. If the user says *today it is hot outside* then the word *hot* is out of vocabulary.

The sentence *warm outside is today it* has no out-of-vocabulary words but is not in the context. This sentences is *out of context*. The term *out of grammar* is often used in the same meaning.

Cerence ASR (Embedded) recognition contexts for list of sentences can be built from a text description called a *grammar*. They are built with the tool grmcpl. These contexts and the grammar are the main topic of this document.

Cerence ASR (Embedded) also still supports so-called *Field contexts*. The are contexts that are built from very large list of items plus structural information. They are built with the tool `fieldcontextcpl`. This type of context is however deprecated and being gradually replaced by solutions based on statistical language models.

Contexts can be combined at runtime. It is possible to combine different types of contexts although not all combinations are possible. Central to the combination of context is the concept of *slot*. We will go into detail on slots later, but imagine a slot being a place-holder inside a sentence of something that is not known up-front. For example: *I want to go from <city> to <city>* intuitively is a context which describes a booking question about cities of which the list of cities is not yet known in advance.

As a general rule a context *A* can be combined with a context *B* if *A* contains one or more slots. You can say that *B* is *connected* to *A* or that *B* is *embedded* inside *A*.

The second approach to context creation is to use a *statistical language models* (SLM). In this case each sentence is given a probability of occurrence. The vocabulary of the context is still finite. So if a user speaks a words that is not in context then the recognition engine will never recognize it. Out of grammar sentences are theoretically not possible because each sentence now has possibility of occurrence. There will be however many sentences that have a probability which is close to 0 and will therefore never be recognized. Statistical model based context can be created with the tool ngramctxcpl or are provided by Cerence. For example, the add-on for messaging dictation provides an SLM based context.

Transcriptions

So far we talked about the terminals inside a context and the sentences they can form. But we still don't know how these terminals are pronounced. To know how a terminal is pronounced you have to associate one or more *phonetic transcriptions* to it.

Phonetic transcriptions are discussed in detail in the Cerence ASR (Embedded) Pronunciation Tools package . Phonetic transcriptions are typically tied to one language. When creating a context from data there are typically 3 sources of phonetic transcriptions:

1. A *dictionary* is essentially a list of terminals along with their transcriptions. Context creation tools can use more than one dictionary. These dictionaries are searched in a particular order. If only dictionaries are used and if a terminal is not in any dictionary then the context cannot be created.
2. The *common linguistic component* is a system that *constructs* phonetic transcriptions from the terminals. The system will do its best to create the best possible transcriptions, but may occasionally come up with an unintended transcription or is some cases with no transcriptions at all (when trying to transcribe a Chinese terminal with an English CLC system for example).
3. Some grammar formats allow phonetic transcriptions inside the grammar. This will be explained further in this document.

Language Models

Starting with Cerence ASR (Embedded) the recognition context can be separated in two parts. The first part, which corresponds to the recognition context as described so far, contains the vocabulary of the task along with the transcription information. This context remains dependent on the acoustic model.

The second part contain the relations among the terminals. This is the **Language Model**. This buffer is not directly dependent on the acoustic model. It is possible to have several vocabulary contexts which correspond to the same language model.

The messaging dictation add-on provides a vocabulary context along with a language model.

Language model buffers are currently only provided by Cerence directly. The is no tool in this package to create them.

grmcpl

This document gives the pre-requisites to run Cerence ASR (Embedded) tools and explains the basic command line syntax. In the rest of this document we will assume that we are starting the tool from the tools directory. If you install your tools on a Windows PC in the default location of the tool then you will find the tools at:

```
c:\\ProgramData\\cerence\\tools\\cerence_asr_embedded_vx_y_z\\ve
```

The easiest way to experiment with tools is to start a tools prompt from the Start Menu which you find at *Cerence > Cerence_ASR_Embedded_Tools_Prompt*.

Let’s assume we’ve created a directory `csdk` in the home direcory. And in this directory we created sub-directories `bnf` and `ctx`. This can done as follows:

```
cd %USERPROFILE%;
mkdir csdk csdk\bnf csdk\ctx
cd csdk
```

Hint:

You can enter the examples in your favorite editor, save them in the `%USERPROFILE%\csdk\bnf` directory, and then run the command line examples while reading the text. We will assume for the rest of this tutorial that we’re running the samples in the directory `bnf`.

Let us now look at our first example of a grammar:

```
#BNF+EMV2.1;
!grammar cdd;
!start <main>;
<main> : !repeat (<digits>, 1, *);
<digits> :  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9;
```

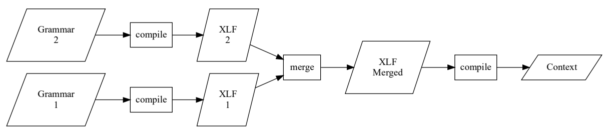
Paste this grammar in your favorite editor and save it in the directory `bnf`. First we will compile this grammar with the tool grmcpl which is started by the following command-line:

```
> grmcpl -g bnf\cdd.bnf -l enu -C ctx\cdd.fcf --bufferSpec=enu -
```

As a result you will find a file `cdd.fcf` in the directory `ctx` which you can use for example in the Companion SDK tool `recog_test`.

The compilation process

Binary contexts can be built from one or more grammar files. BNF+ and JSGF grammars are used to build *intermediate files* which use a file format called XLF (eXtensible Language Format). A context is created from an XLF file. One or more XLF files can be merged together. A diagram of the organization of the compilation process (grammar, XLF files, and binary context) is shown in the following figure:



We will now continue to explain the use of the Cerence BNF+ grammar format. The BNF+ format is historically the native format for Cerence ASR (Embedded) engine, this format has most features and is the preferred format to use if you start a new application.

It is preferred to encode grammars in UTF-8 [1]

[1] Many tools, including *grmcpl*, also support UTF-16 encoding, but the Cerence ASR (Embedded) API supports only UTF-8. We recommend therefore to use UTF-8.

Verifying the coverage of your grammar

As grammars become larger it becomes too hard to check whether your grammar covers the sentences you intend it to recognize. Also, your grammar may recognize sentences that should not be recognized at all.

To help testing grammar coverage we have the tool **ctxverify**. Let's take our example grammar and make `ctxverify` generate some sentences:

```
> ctxverify -g bnf\cdd.bnf --mode=generate --maxSentLen=3
```

This will generate output like the following:

```
0
0 0
0 0 0
0 0 1
0 0 2
...
```

You can see all the accepted utterances for this particular grammar which consist of at most 3 words. You can also check whether a particular sentence is in the grammar:

```
ctxverify -g bnf\cdd.bnf --mode=check
```

Will give you a prompt. Try entering *1 2 3 4 5* and you will get:

```
>ctxverify.py -g bnf\cdd.bnf -m check
Compilation of grammar 'bnf\cdd.bnf' to xlf succeeded
Type a sentence after '>', then press 'return' to check the sent
To stop checking type Ctrl+Z and press 'return'
> 1 2 3 4 5
IN:    1 2 3 4 5
UNIQ:  1 2 3 4 5
```

The Cerence BNF+ grammar format

Basic Grammar Concepts

BNF+ is based on the concept of Context Free Grammars (CFGs). The core of a context free grammar is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. An example of a grammar rule in BNF+ might be as follows:

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

`<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

BNF+ uses a C-like syntax:

- Grammars are composed of statements terminated by a semicolon: `;`.
- Comments can be line comments which start with `//` or block comments where the comment text is surrounded by `/*` and `*/`.
- Parentheses `(` and `)` can be used to group things together.

Statements, rules, and symbols in BNF+

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

```
#BNF+EMV2.1;
!grammar Drinks;
!start <Speech>;

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

You can compile this grammar, which is assumed to be saved as *drinks.bnf*, into a recognition context with the following command line:

```
> grmcpl -g bnf\drinks.bnf -l enu -C ctx\drinks.fcf --bufferSpec
```

The resulting context file `drinks.fcf` can be loaded in the `recog_test` tools of the companion SDK.

This grammar allows users to order either a lemonade, a milkshake, or an orange juice. The first two lines of the grammar start with an exclamation mark (`!`) followed by a word. These words are directives, which means they are words that have special meaning to the grammar compiler.

The grammar contains non-terminals, which, as previously described, represent rules. The name of a non-terminal can be any combination of almost any character from the unicode character set, excluding the “greater than” symbol (`>`) and the “less than” symbol (`<`).

The hash or pound (`#`) symbol has a special meaning which is explained in the section Importing and exporting rules in BNF+.

The name of the non-terminal is enclosed by angle brackets (`< >`). BNF+ has 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 written in 2 ways. The most straightforward way is to write it enclosed by double quotes, for example `"The Recognition Bar"`. You can write pretty much any sequence of printable characters between quotes. You can also omit the quotes but then the set of allowed characters is restricted to script characters. Obviously you can also not write non-quoted terminals with white-space inside them, because the white-space indicates where a non-quoted terminal starts and ends. The specification section at the end of this document specifies exactly which characters can be used inside quoted and non-quoted terminals.

All grammars start with a header; for BNF+ grammars the header is `#BNF+EMV2.1;` The header is required and must be on the first line of the file. The headers may not be preceded by any character including white space; the only exception is the UTF-8 Byte Order Mark (`0xef 0xbf 0xbe`) which is ignored by the compiler.

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. It ends with a semicolon (`;`).

Grammar statements are required. The combination of a grammar name and a rule name is used to uniquely identify the rules when using multiple grammars. Such a combined rule is called a *fully qualified rule*. In the example above the fully qualified name of the rule `<Speech>` is

`Drinks#Speech`. In the Cerence ASR (Embedded) API you will have to refer to rules by their fully qualified name. Note that the angled brackets are not part of the rule name!

The second statement in the example 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>`.

Due to the start statement `<Speech>` is a start rule. Since there is only one start rule the conclusion is that only the three drinks defined in body of `<Speech>` 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+EMV2.1;
/* This grammar allows the user to order a hamburger and fries i
   to the drinks. */
!grammar Order;
!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 and exporting rules in BNF+) 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 and exporting rules in BNF+). Therefore, the same private rule name can be re-used in several grammars that are merged into one context, without introducing conflicts.

- A combination of rules, terminals, pipes, parentheses and other operators is called an *expression*. For example `(a | (<b> | "c"))` is an expression, `(<b> | "c")` is also an expression. The exact definition of an expression can be found in the section BNF+EMV2.1 Syntax at the end of this document.

Let us revisit the compilation command line:

```
> grmcpl -g bnf\drinks.bnf -l enu -C ctx\drinks.fcf --bufferSpec
```

Why is the `--bufferSpec` option needed? This option tells the compiler to use the CLC component for American English (ENU) to look up phonetic transcriptions for the terminals in the grammar. If we would not add this option then the compiler would tell us the it has no transcriptions for all terminals in the grammar.

Optional recognition

Optional Recognition in BNF+

Assume that we would like to add some politeness statements to the grammar. We would do this to cover better what the user may say. Let’s say we allow the user to say “please” at the end of the command, but we don’t want to make the use of “please” required. We would write a grammar like the following to express the fact that “please” is optional:

```
#BNF+EMV2.1;
!grammar polite_drinks;
!start <Order>;

<Order>: <Drink> !optional(please);
<Drink>: a lemonade | a milkshake | an orange juice;
```

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

```
#BNF+EMV2.1;
!grammar polite_drinks_long;
!start <Order>;

<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:

- Any expression can be made optional.
- Square braces are a convenient shorthand for the directive `!optional`: `!optional(x)` can also be written as `[x]`.

Repeated recognition

Repeat in BNF+

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

```
#BNF+EMV2.1;
!grammar 3_drinks;
!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+EMV2.1;
!grammar 1_to_3_drinks;
!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+EMV2.1;
!grammar 1_to_10_drinks;
!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, you can also specify this number rather than using the star. This avoids having to check in the result processing code whether more than the maximum number of repetitions occurred. A finite number of repetitions usually leads to bigger recognition contexts.
- 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. In this case the repeated expression is also made optional.

The first argument can be any expression (a formal definition of BNF+ 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,*)`.

The number of arguments in `!repeat` can be reduced:

`!repeat(expression,*)` is a shorthand for `!repeat(expression,0,*)`.
`!repeat(expression,+)` is a shorthand for `!repeat(expression,1,*)`.

Examples of a standalone BNF+ grammar

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. In BNF+ this grammar becomes:

```
#BNF+EMV2.1;
!grammar a_lot_of_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 | 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 (because `please` is not optional)
- one lemonades please (because the plural form `lemonades` can only follow `<Count>`).

Recursion

Warning:

Recursion is not allowed in BNF+.

Referencing a rule in its own definition (right hand side) is called recursion. Recursion can be direct, when rule references itself in its definition, or indirect, when there is a cyclic dependency between rules.

Example of direct recursion in BNF+:

```
<GNU> : <GNU> is Not Linux;
```

Example of indirect recursion in BNF+:

```
<A> : bear who says <QUOTE>;
```

```
<QUOTE> : I am <A>;
```

Combining Grammars

Importing and exporting rules in BNF+

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` statement. 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:

snack_shop.bnf

```
#BNF+EMV2.1;
!grammar snack_shop;
!export <Food>;

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

bar.bnf

```
#BNF+EMV2.1;
!grammar bar;
!start <Order>;
!import <snack_shop#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.

This example can be compiled with the following command:

```
> grmcpl bnf\snack_shop.bnf,bnf\bar.bnf -l enu -C ctx\bar.fcf --
```

The compiler figures out automatically that `bar.bnf` uses `snack_shop.bnf`.

Therefore `bar#Order` is the only start rule in the resulting context.

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.
- If a grammar with a start rule is used inside another grammar then the `!start` of the imported grammar is ignored.
- If 2 grammars a compiled into one context and if they do not use rules from the other then the start rules from both grammars are visible in the final context.
- You need to specify from which grammar you want to import the rule. The grammar name acts as a name space for rules. This is why grammar names are required.
- 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.
- Circular imports are not allowed. Each of the grammars in the following example is legal as a standalone grammar but they will not compile together because they import rules from each other:

  snack_shop.bnf

  ```
  #BNF+EMV2.1;
  !grammar snack_shop;
  !import <bar#Drink>;
  !export <Food>;

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

  bar.bnf

  ```
  #BNF+EMV2.1;
  !grammar bar;
  !start <Order>;
  ```

```
!import <snack_shop#Food>;
!export <Drink>;

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

- Imports are resolved during the merging step of the context compilation process. All grammars must be present at that moment if you want to create a recognition context.

Pronunciations

The pronunciation of words used in a context can be specified in several different ways in Cerence BNF+ grammars. By default, the words used in the grammar are passed to a grapheme-to-phoneme expert system (CLC), which converts them to sequences of phonemes. Alternatively, a dictionary of phonetic transcriptions can be used. If you want to use a dictionary, you must specify the dictionaries when calling the grammar compiler. Words for which there are transcriptions in the dictionary will not be passed to the CLC expert system. Pronunciations can also be specified directly in the grammar via `!pronounce` statements or `!pronounce` directives. Pronunciations inside grammars take precedence over transcriptions specified in dictionaries or supplied by the CLC expert system. The next two sections describe `!pronounce` statement and `!pronounce` directive in more detail.

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

1. `!pronounce` directives in the grammar.
2. `!pronounce` statement in the grammar.
3. Transcriptions supplied in the dictionary passes in option `--clcOverrideDictionaryFilepath` used when compiling the grammar.
4. Transcriptions provided automatically by the CLC component, possibly with the assistance of user dictionaries passed in `--userDictionaryFilepaths`

A test dictionary

In the directory `dct` create the following dictionary:

```
[Header]
Language=enu
```

```
[SubHeader]
Content=EDCT_CONTENT_BROAD_NARROWS
Representation=EDCT_REPR_SZZ_STRING

[Data]
coffee // "#'kO.fi#"
coffee // "#'kA.fi#"
```

We have to compile this text dictionary into a binary dictionary. This is done with the command:

```
    > dictcpl -i dct\bar.dct -o dct\bar.dcb
```

In the `bnf` directory we modify `bar.bnf` by adding the drink `coffee`:

bar.bnf

```
#BNF+EMV2.1;
!grammar bar;
!start <Drink>;

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

Compilation options

If you want to use one or more dictionaries for compilation you use the `-d` or `--userDictionaryFilepaths` option:

```
> grmcpl bnf\bar.bnf -d dct\bar.dcb -l enu -C ctx\bar.fcf --buff
```

This `--userDictionaryFilepaths` option will instruct the CLC component that it has to use user-specified transcriptions for the word `coffee`. Please refer to the documentation of the pronunciation tools for more information on pronunciation generation.

You can use multiple dictionaries as in:

```
> grmcpl -g my_grammar.bnf -C my_grammar.fcf -d my_dict_1.dct;my
```

In this case the transcription for a terminal `t` will be chosen from the left-most dictionary on the command-line which contain a transcription for `t`. In other words: if both *my_dict_1.dct* and *my_dict_2.dcb* contain transcriptions for the terminal `t` then the transcriptions from *my_dict_1.dct* will be used and those in the second grammar will be unused.

Grammar-wide pronounces

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 transcriptions 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 CLC 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 transcriptions are desired, the `!pronounce` directive must be used. The following sample grammar provides a possible solution:

```
#BNF+EMV2.1;
!grammar bar;
!start <Drink>;
!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).

An interesting application of PRONAS is for Mandarin Chinese. Here we can use `PRONAS` to transibe hànzì symbols using pinyin instead of phonetic transcriptions:

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

If we had use an L&H+ transcription we we will have to write:

```
    !pronounce 中国  "t&s+o55nK.kwO35";  /* L&H+ */
```

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 CLC; in the second case, the CLC is overruled by the phonetic transcription in the grammar.

The same argument holds for Japanese where the pronunciation of Kanji characters can be done by using hiragna or katakana with PRONAS.

Word-specific pronounces

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:

```
#BNF+EMV2.1;
!grammar dif_pron;
!start <difPron>;

!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;
```

Note:

The `!pronounce` directive follows the terminal for which it specifies the transcription. Alternative transcription types and alternative transcriptions can be specified in the same way as in the pronounce statement.

The transcriptions that are specified with a !pronounce directive on a specific instance of a word supersede the transcriptions that are specified for that word with a pronounce statement.

Character set and CLC

Grammars must be written in the UTF-8 encoding of Unicode. When passing text data to the Cerence ASR (Embedded) 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 CLC expert system is called to generate transcriptions. Since the CLC 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 CLC module will not interpret Cyrillic characters correctly. If dictionaries or pronounce statements are used, the character set does not have these additional limitations.

User IDs in BNF+

The `!id` directive allows the user to specify a User ID for a terminal. The User ID must be an unsigned 64-bit 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(0x00000001) | initiate !id(aAQ==)
```

As the same User 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. User IDs never appear in the recognized string.

User IDs have a default ID of $2^{64} - 1 = 0\mathrm{xffffffffffffffff}$.

As seen in the example user IDs can be written in decimal (1), hexadecimal and in base64 form.

Functional note:

> User IDs can be found in the JSON result string. Refer to the Cerence ASR (Embedded) Runtime documentation for more information.

Runtime modification of grammars

Modification in BNF+

Sometimes the entire specification of the grammar is not known until run-time. In this event, the grammar needs to be modifiable, or, in other words, some of its rules need to be modifiable.

Cerence ASR (Embedded) allows to attach several types of context to a modifiable rule. Rules can be modified only if the `!slot` statement is used. Following grammars is an example of the use of the !slot statement:

```
#BNF+EMV2.1;
!grammar modifiable_grammar;
!slot <names>;
!start <phone dialer>;

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

Before the addition of a context with names to the context which is compiled from this grammar, this grammar can never return a result. However, because the `!slot` statement was used, a context can be attached to the `<names>` non-terminal at runtime. The modification of the compiled context happens with the `connection` keys in the JSON specifications. Important to remember is that you have to refer to the name of the slot with its full name: `modifiable_grammar#names`!

You cannot redefine slot rules in the same grammar. The following grammar is illegal:

```
#BNF+EMV2.1;
!grammar modifiable_grammar;
!slot <names>;
!start <phone dialer>;
```

```
    <phone dialer>: !optional(Please) call <names>;
    <names>: John Doe; // Illegal!! <names> cannot be defined: it's
```

Tags in BNF+: A simple semantic annotation

Tags are a mechanism to identify (tag) parts of the recognized sentence. We first explain the reason for introducing tags. Then we introduce the `!tag` directive and its syntax.

Silence terminals as markers

As we saw in previous section user IDs are linked to terminals. However we often don't really care to which terminal a user ID is associated.

For example, we could use user IDs as a mechanism to indicate sentence properties. We don't care where the user ID is located as long as it is found somewhere in the sequence of output terminals. An often used technique to insert user IDs at arbitrary locations in the grammar is to define a terminal with transcription silence and give this terminal a user ID:

```
    #BNF+EMV2.1;
    !grammar silence_with_id;
    !start <start>;
    !pronounce _ "##";

    <start>: this is a sentence with a identifier _ !id(1234);
```

User IDs are also used to identify parts of sentences in the output. This is done by sandwiching a part of a grammar, for instance a rule, between silence terminals with user IDs:

```
    #BNF+EMV2.1;
    !grammar expression_with_silences;
    !start <start>;
    !pronounce _ "##";

    <start>: this is a <rule>;
    <rule>: _ !id(1234) (hot dog | burger) _ !id(1234);
```

In the result this leads to two instances of the terminal _. These instances are 'real' terminals because they have a different begin and end time.

Slots as markers

Slots also appear in the result in a way that marks the location of the slot in the utterance. Therefore users sometimes use slots as an NLU mechanism without having need for the extension properties of the slot:

```
#BNF+EMV2.1;
!grammar slot_for_tagging;
!start <start>;
!slot <rule>;

<start>: this is a <rule>;
```

Using a slot this way complicates matters for the user.

Tagging of expressions

Silence nodes are a trick which should be used with care. One silence node often has little effect on accuracy, The issue is that if we use too many of them it *does* effect performance. For larger projects we recommend to switch to Statistical Language Models with a sem3 rule-based NLU grammar.

The `!tag` statement allows us to insert strings and/or user IDs without requiring something to be recognized. A *tag* can be associated with any expression. The tag can identified with a string, a user ID or both:

```
#BNF+EMV2.1;
!grammar string_and_id_on_expression;
!start <start>;

<start>: a <rule> please;
<rule>: !tag(FOOD!id(9999), hot dog!id(111) | burger!id(222));
```

With only a user ID:

```
#BNF+EMV2.1;
!grammar id_on_expression;
!start <start>;

<start>: a <rule> please;
<rule>: !tag(!id(9999), hot dog!id(111) | burger!id(222));
```

With only a string:

```
#BNF+EMV2.1;
!grammar string_on_expression;
!start <start>;
```

```
<start>: a <rule> please;
<rule>: !tag(FOOD, hot dog | burger);
```

As we will see in the section on results the output of tag statement is very similar to that of the slot statement.

Note that the string inside a tag is not a terminal, it does not require transcriptions. from the result point of view a tag behaves more like a rule than a terminal.

Empty tags

It may be useful to insert a single tag, not associated to any expression or terminal. This can be done as follows:

```
#BNF+EMV2.1;
!grammar isolated_tag;
!start <start>;

<start>: this is a !tag(TAG!id(1234),<NULL>);
```

Refer to the section Null Rule for a more complete description of the null rule.

Tags in the result

In the result tags are also fully qualified with the grammar name, just like rule names. For example: “this is a hot dog” will produce the following JSON result. (We omit confidences, scores and timings):

```
{ "resultType": "NBest",
  "hypotheses": [
    { "startRule":"string_and_id_on_expression#start",
      "items": [
        {  "type": "terminal",
           "orthography": "a",
        },
        {  "type": "tag",
           "name": "string_and_id_on_expression#food",
           "userID.hi32":0,
           "userID.lo32":9999,
           "items": [
             {  "type": "terminal",
                "orthography": "hot"
             },
             {  "type": "terminal",
                "userID.hi32":0,
                "userID.lo32":111,
                "orthography": "dog"
             }
           ]
        },
        {  "type": "terminal",
           "orthography":"please"
        }
    ]
  }
}
```

For the empty tag the result will be:

```
  { "resultType": "NBest",
    "hypotheses": [
      { "startRule":"string_and_id_on_expression#start",
        "items": [
          {  "type": "terminal",
             "orthography": "this"
          },
          {  "type": "terminal",
             "orthography": "is"
          },
          {  "type": "terminal",
             "orthography": "a"
          },
          {  "type": "tag",
             "name": "string_and_id_on_expression#TAG",
             "userID.hi32":0,
             "userID.lo32":1234
          },
      ]
    }
  }
```

Relation between slots and tags

Slots can been seen as an union of 2 orthogonal features:

1. The capability to dynamically attach another context to a context.
2. Identification of parts of the grammar in the result; which is *the same as tags*.

From this follows that is logical to represent slots and tags identically in the output. In fact, slots can be viewed as automatic tags. The tag of a slot is its rule name.

Pre-filled slots and tags

A fully qualified rule name `grammar#rule` may also be a tag name. We do not prevent this as this does not cause problems. On the contrary this property can used to achieve the effect of a pre-filled slots with FST contexts:

```
#BNF+EMV2.1;
!grammar prefilled_slots;
!start <start>;
!slot <rule>;

<start>: a (<rule>|!tag(rule, pre-filled content)) please;
```

Rule Activation

There may be times when you want to recognize only certain rules in a grammar. Take for example 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+EMV2.1;
!grammar activatable_example;
!start <top-rule>;
!activatable <wake-up>;
!activatable <rest-of-grammar>;

<top-rule> : <wake-up> | <rest-of-grammar>;
<rest-of-grammar>: send e-mail | start web browser | [go to] sle
<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>`.

Notes:

- Start-rules are by definition activatable.

- The default state of a activatable rule is “active”. When a context is created in the Cerence ASR (Embedded) 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.

Special rules and terminals

BNF+ has special two special rules: `<NULL>` and `<VOID>`. These rules are universally defined and therefore don’t have to be imported. They cannot be redefined.

There is also a construct to match “any speech”. In BNF+ this is done with a reserved rule `<...>`.

Garbage or Any Speech

Garbage or any speech is way of absorbing parts of speech which are not known in advance.

The use of the garbage model should be limited. Refer to the description of the garbage penalty parameter `LH_REC_PARAM_GARBAGE` in the Cerence ASR (Embedded) API Reference for more information.

Garbage in BNF+

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 rule `<...>`, which models any speech. Here is an example of its use:

```
#BNF+EMV2.1;
!grammar order_with_garbage;
!start <Order>

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

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 in the recognition result, but the user may have said: *I want a lemonade please*. The first instance of `<...>` matches *I want a*, the second matches *please*.

Null and Void Rules

Null Rule

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

```
<WithNull> : an <NULL> orange juice <NULL> please // BNF+;
```

are exactly equivalent in recognition to the rules:

```
<WithNull> : an orange juice please // BNF+;
```

Note:

- In BNF+ `!optional(x)` has the same effect as `(x | <NULL>)`.

Void Rule

`<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 rules:

```
<WithVoid> : an <VOID> orange juice please; // BNF+
```

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:

```
/* illegal BNF+ rules */
<rulename> : ;
<another rule> : word | ;
```

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

```
/* Legal BNF+ */
<rulename2> : <NULL>;
<rulename3> : <VOID>;
<rulename4> : word | <NULL>;
```

`<NULL>` rules can be useful inside an expression surrounded by `!tag`. Refer to Tags in BNF+: A simple semantic annotation for more information.

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.

Double quoted terminals with spaces

Spaces inside double quoted terminals should be used with caution. The use of quotes 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 between double quotes will appear in the recognition result as a single word , 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 may not be able to 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. The FST search algorithm is less sensitive to this phenomenon than the older LexTreeDP and TreeDP algorithms.

Spaces inside double quotes have an effect on the recognition and the memory needs of the system. The phonetic transcriptions of double quoted multi-word alternatives will usually contain underscores. E.g. the phonetic transcription for “Bill Clinton” in ENU is `#'bIl_'klIn.t$n#`. Such transcriptions are internally converted to two alternative transcriptions: one in which all the underscores are modeled as silence and one in which all the underscores are removed. This doubling of transcriptions has an impact on the memory and CPU needs of the application. [2]

[2] One of the alternative transcriptions 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 CLC 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.

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, be careful with repetitions. If you know the maximum number of repetitions, it *may* be 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. Keep in mind 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 again that you use an infinite maximum number because of memory considerations. Remeber that your result processing code then has to filter out results with more than 20 digits!

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 without optimization the context created from the first line would have more terminals than that of the second line. Context optimization will remove the redundant terminals.

Note that `!id` statements may interfere with optimization. Changing the previous example to the following example will block optimization:

```
what's !id(100) your name | what's your address;
what's !id(101) your (name | address);
```

Due to the different word ids the 2 instances of `what's` are no longer equivalent. Pronounce directives may have the same effect:

```
what's !pronounce("#'wAts#") your name | what's your address;
what's !pronounce("#'huz#") your (name | address);
```

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.

Formal Specifications

Token Specification Format

The grammar is specified using the ABNF notation. [RFC2234] ABNF is an internet standard for specifying the syntax of all kinds of text by means of context-free formalism.

If we want to use characters as literals then we write them between single quotes as in `"("`. The double quote itself is usually written as `DQUOTE`.

A token is specified by a rule. The following rule specifies a token that consists of the letters *t*, *o*, *k* , *e* and *n*:

```
MY-TOKEN-NAME = "t" "o" "k" "e" "n"
```

As a convention we use uppercase letters in token names.

Letters can be grouped as in:

```
MY-TOKEN-NAME = "t" ("o" "k" "e") "n"
```

You can introduce alternatives with the forward slash (`/`) symbol. The following rule says that *taken* is also a valid alternative for the *MY-TOKEN-NAME* token:

```
MY-TOKEN-NAME = "t" ("o" / "a") "k" "e" "n"
```

Repeating parts of an expression is also possible. The following rule says that *token*, *token_*, *token__*,… are alternatives for the token:

```
MY-TOKEN-NAME = "t" "o" "k" "e" "n" *"_"
```

Alternatively `1*` means repeat 1 or more times, `*1` means *this part is optional*, `2*4` means *this part is repeated between 2 and 4 times* and `3*3` means *this part is repeated exactly 3 times*. For example:

```
MY-TOKEN-NAME = "t" "o" "k" "e" "n" 2*4"_"
```

Sets of characters can be specified with a special syntax which uses a character set code point. We will use the Unicode charactr set. This is useful because Unicode often groups related symbols together. If we want to specify a token that my only consist of **letters** but not the symbols of the Latin-1 Supplement table then we write:

```
MY-TOKEN-NAME = %x00c0-00ff
```

You can add individual Unicode code points as follows:

```
MY-TOKEN-NAME = %x0020 / %x00c0-00ff
```

The previous example adds the space character to the allowed character set.

[RFC2234] Augmented BNF for Syntax Specifications: ABNF (http://www.ietf.org/rfc/rfc2234.txt)

BNF+EMV2.1 Formal Specification

BNF+ aims as much as possible of the Unicode character set. The problem is that “supporting the unicode character set” is a far reaching statement. We try to allow pretty much everything inside quotes except:

i. Control characters (U+0000-U+001F,U+007F-U+009F)
ii. Surrogates (U+D800-U+DFFF)
iii. Replacement characters and byte-order marks (U+FFFD,U+FFFE,U+FFFF).

For unquoted string we limited ourselves to a number of alphabetic scripts, supplemented by a broad range of Chinese, Japanese and Korean code points.

You will find the following notation `UNICHR-XX`. The symbols `XX` indicate a general Unicode category value [UnicodeTR44]. For example `UNICHR-N` denotes any unicode character with the number property.

We also used an extended notation `X - Y` to indicate all characters in `X` minus those in `Y`.

[UnicodeTR44]	Unicode Character Database: Property Values (http://www.unicode.org/reports/tr44/#Property_Values)

BNF+ Tokens

The following rules define common parts of token but are not recognized as individual tokens:

```
NEW-LINE = %x000d %x000a / %x000a / %x000d ; \r\n or \n or \r

DQUOTE = %x0022 ; "

WHITE-SPACE = %x0009 / %x000d / %x000a / %x0020 / %x00a0
               ; White space is generally, but not always, ignor

SIMPLE-SPACE = %x0009 / %x0020 / %x00a0
               ; Just white space between letters, no tabs or ne

ESCAPE-SEQUENCE = "\" ("n" / "r" / "t" / "v" / "\" / DQUOTE)

DIGIT = "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9

HEX-DIGIT =  DIGIT / "a" / "A" / "b" / "B" / "c" / "C" / "d" / "
```

```
    BASE64-SYMBOL = %x0041-005a /    ; A-Z
                    %x0061-007a /    ; a-z
                    "+" / "/" / "="

    UNICODE-CHAR = "\" "u" 4*4HEX-DIGIT  ; example ÿ -> \u00ff

    QUOTED-STRING-CHAR = (UNICHR-L / UNICHR-M / UNICHR-N / UNICHR-P
                         -  (%x0022 / %x005c);  without \ and " and

    NON-TERMINAL-CHAR = %x0020-003b / %x003d / %x003f-%x007e / %x00a
                          ; ASCII + Latin-1 without control chars an

    TERMINAL-LETTER = "'" / "&" / "." / ; Common punctuation inside
                      UNICHR-L / UNICHR-M / UNICHR-N / UNICHR-PC / U
```

The following rules specify the tokens of BNF+:

```
    QUOTED-STRING = DQUOTE *(QUOTED-STRING-CHAR / ESCAPE-SEQUENCE /
                         ; A quoted string can contain any character
                         ;The characters " and \ must be escaped insi


    NON-TERMINAL = "<" 1*NON-TERMINAL-CHAR ">"
                        ; A non-terminal can contain any character th
                        ; The exception is the use of < and >.
                        ; There is no escape mechanism for these char
                        ; We feel this is not a severe limitation.

    TERMINAL = *(TERMINAL-LETTER / DIGIT) TERMINAL-LETTER *(TERMINAL
                  ; A terminal primarily consists of alphanumeric cha
                  ; A number of word-internal punctuation characters

    INTEGER = 1*DIGIT /
              "0x" 1*HEX-DIGIT /
              ("a" / "A") "6" "4" 1*BASE64-SYMBOL
                ; Integers are not part of this terminal token defin
                ; BNF+ uses integers for 2 functions: as terminals b
                ; Therefore we need to distinguish integers from oth
                ; Integers can be used as terminals too.

    GRAMMAR-KEYWORD = "!grammar"

    LANGUAGE-KEYWORD = "!language"

    START-KEYWORD = "!start"

    EXPORT-KEYWORD = "!export"

    IMPORT-KEYWORD = "!import"
```

```
SLOT-KEYWORD = "!slot"
                 ; !slot was originally proposed as a synonym fo

ACTIVATABLE-KEYWORD = "!activatable"

PRONOUNCE-KEYWORD = "!pronounce"

PRONOUNCE-OPEN = "!pronounce" *SIMPLE-SPACE "("

OPTIONAL-OPEN = "!optional" *SIMPLE-SPACE "("

ID-OPEN = "!id" *SIMPLE-SPACE "("

REPEAT-OPEN = "!repeat" *SIMPLE-SPACE "("

TAG-OPEN = "!tag" *SIMPLE-SPACE "("

OPEN-SQUARE-BRACKET = "["

CLOSE-SQUARE-BRACKET = "]"

OPEN-PARENTHESIS = "("

CLOSE-PARENTHESIS = ")"

COLON = ":"

PIPE = "|"

PLUS = "+"

STAR = "*"

COMMA = ","

TERMINATOR = ";"
```

Self-Identifying Header

The first characters of the file must contain the header. The header is specified by:

```
HEADER = "#" "B" "N" "F" "+" "E" "M" *WHITE-SPACE "V" 1*DIGIT ".
```

The following version numbers should be accepted:

- 1.0
- 1.1

- 2.0
- 2.1

Comments

Comments are ignored. Comments are defined by:

```
LINE-COMMENT-CHAR = SIMPLE-SPACE /
                    %x0020-%x007e /
                    %x00a0-%x00ff

LINE-COMMENT = "//" *LINE-COMMENT-CHAR

BLOCK-COMMENT-CHAR = WHITE-SPACE /
                     %x0020-%x0029 /
                     %x002b-%x002e /
                     %x0030-%x007e /
                     %x00a0-%x00ff

BLOCK-COMMENT = "/*" *((*"*" / *"/") *BLOCK-COMMENT-CHAR) "*/"
                   ; Difficult way of saying: */ is not allowed
```

BNF+EMV2.1 Syntax

This is the syntax of BNF+ in ABNF:

```
grammar = *statement

statement = grammar-statement /
            export-statement /
            import-statement /
            slot-statement /
            grammar-statement /
            activable-statement /
            pronounce-statement /
            language-statement /
            rule

grammar-statement = GRAMMAR-KEYWORD (QUOTED-STRING / TERMINAL) T

start-statement = START-KEYWORD rule-list TERMINATOR

export-statement = EXPORT-KEYWORD rule-list TERMINATOR

import-statement = IMPORT-KEYWORD rule-list TERMINATOR

slot-statement = SLOT-KEYWORD rule-list TERMINATOR
```

```
activatable-statement = IMPORT-KEYWORD rule-list TERMINATOR

pronounce-statement = PRONOUNCE-KEYWORD (TERMINAL / QUOTED-STRIN

language-statement = LANGUAGE-KEYWORD (TERMINAL / QUOTED-STRING)
                      ; Included for backwards compatibility
                      ; Has no meaning and may even trigger a c

rule-list = 1*NON-TERMINAL

transcription-list = transcription *(PIPE transcription)

transcription = *1(TERMINAL) QUOTED-STRING

rule = NON-TERMINAL COLON expression TERMINATOR

expression = alternative *(PIPE expression) *(PLUS / STAR)

alternative = factor *(factor)

factor = (TERMINAL / QUOTED-STRING / INTEGER) *(terminal-modifie
         NON-TERMINAL /
         OPEN-PARENTHESIS expression CLOSE-PARENTHESIS /
         OPEN-SQUARE-BRACKET expression CLOSE-SQUARE-BRACKET /
         OPTIONAL-OPEN expression CLOSE-PARENTHESIS /
         REPEAT-OPEN expression COMMA repeat-body CLOSE-PARENTHE
         TAG-OPEN tag COMMA expression CLOSE-PARENTHESIS

repeat-body = INTEGER ( COMMA (INTEGER / STAR) / STAR / PLUS )

terminal-modifier = ID-OPEN INTEGER CLOSE-PARENTHESIS /
                    PRONOUNCE-OPEN transcription-list CLOSE-PARE

tag = TERMINAL *1(ID-OPEN INTEGER CLOSE-PARENTHESIS) /
      ID-OPEN INTEGER CLOSE-PARENTHESIS
```