Vestec Automatic Speech Recognition Engine
Standard Edition
Version 1.1.1

Grammar Developer's Guide

Vestec Automatic Speech Recognition Engine
Standard Edition
Version 1.1.1
Grammar Developer's Guide

145 Columbia Street West, Suite 1, Waterloo, Ontario, Canada N2L 3L2

Table of Contents

About This Document

Vestec's Automatic Speech Recognition Engine (VASRE) is a speaker-independent speech recognition engine that supports a distributed architecture of servers and clients. VASRE works for Windows, GNU/Linux, and Open Solaris platforms to process an audio file or stream from external sources, such as telephone systems. The grammar can be built simply, with a list of keywords or pronunciations to be recognized, or with a more sophisticated industry standard.

This guide explains the syntax of VASRE text grammars, which is based on Speech Recognition Grammar Specification (SRSG) version 1.0 and Semantic Interpretation for Speech Recognition (SISR) version 1.0.

Audience

This guide is intended for speech application developers who are writing text grammars for specific recognition purposes.

Organization

This guide is organized as follows:

- Section 1 outlines the basic structure of text grammars.
- Section 2 explains the syntax of grammar header.
- Section 3 explains the definition of grammar rules.
- Section 4 presents the format of rule names and tokens.
- Section 5 explains how white spaces, tabs, and new lines work as field delimiters and how the case of grammar text matters.
- Section 6 explains how to add comments to text grammars.
- Section 7 describes how to specify the pronunciations of user-defined tokens.
- Section 8 explains the syntax of alternative operators.
- Section 9 explains the syntax of priority parentheses.
- Section 10 explains the syntax of option brackets.
- Section 11 explains the syntax of repetition brackets.
- Section 12 introduces special rules predefined for special recognition purposes.
- Section 13 explains how to append tags to tokens or sub-rules of text grammars.
- Section 14 explains how the vocabulary size of a text grammar is counted.
- Section 15 introduces sample grammars.
- Appendix: List of Text Grammar Error Codes lists the error codes related to the syntax of text grammars.

Conventions

Guides for VASRE use the following conventions:

- **Bold Arial** represents user utterances, recognized strings, and semantic results.
- `Courier New` represents file names, directory names, command line strings, and file contents.
- *Italic text* represents types, tokens, keywords, variables, and functions.
- Underlined text represents menu strings or texts in graphical user interface.
- *`Italic Courier New`* represents values replaced by you. For example, *`YYYY-MM-DD`* may represent a date in the year-month-day format.
- A paragraph starting with **N.B.** represents critical information or warning.
- For abbreviated terms, both singular and plural are spelled the same. For example, RM represents both resource manager and resource managers.

1 Basic Structure

The format of the VASRE grammar can be either text or binary. A grammar developer starts grammar development by writing a text grammar. The text grammar describes which words, phrases, or sentences should be matched to the input speech. If the text grammar is syntactically correct, the binary grammar can be generated via the compilation of the text grammar. Both text and binary grammars can be added to the server; however, it is a better strategy to add binary grammars because adding text grammars forces the server to compile them, which will consequently degrade the recognition speed.

VASRE supports the syntax of text grammar based on SRGS version 1.0 and SISR version 1.0, which are W3C standards. Visit W3C website for more details on SRGS and SISR. The VASRE Standard Edition (SE) contains eight sample text grammars for your reference. See Section 15 or check the grammar files in `/opt/VestecASRE/Samples/Grammars/` directory for GNU/Linux and `VestecASRE\Samples\Grammars\` for Windows.

The text grammar comprises two parts: header and rule definitions. The header has one or more lines declaring grammar version, language, root rule, and tag formats while each of the rule definitions describes which combination of words or sub-rules the rule should be matched to.

Consider the following example:

```
#ABNF 1.0;             // header line 1
root $Yesno;          // header line 2
$Yesno = $Yes | $No; // rule definition 1
$Yes = yes [please]; // rule definition 2
$No = no [thanks];   // rule definition 3
```

The first two lines comprise the header while the last three lines are rule definitions.

Header declarations and rule definitions cannot be combined. That is, a header declaration must not lie amongst rule definitions.

2 Header

The header must start with a self-identifying header shown below:

```
#ABNF 1.0;
```

The number followed by the semi-colon represents the ABNF version number. Only version 1.0 is supported at this point. You may optionally specify character encoding after the version number. Only *US-ASCII* is supported as illustrated below:

```
#ABNF 1.0 US-ASCII;
```

Root rule declaration may follow the self-identifying header. It is highly recommended to declare the root rule, which acts as the entry point of speech recognition. The root rule declaration is composed of the keyword *root*, the root rule name, and a semicolon. If the root rule is not declared, the root rule is automatically generated as the alternative of all the rules defined in the grammar. In the following example, the root rule will be internally generated as an alternative of *$Yes* and *$No* and hence matches **yes**, **yes please**, **no**, and **no thanks**.

```
#ABNF 1.0;
$Yes = yes [please];
$No = no [thanks];
```

Within the grammar header, you may also declare language, mode, and tag format. For language, you can declare either of two identifiers, *en* or *en-us*, which identically represent US English. For mode, you can declare *voice* only. For tag format, you can declare <*semantics/1.0*>, <*semantics/1.0.2006*>, <*semantics/1.0-literals*> or <*semantics/1.0.2006-literals*>. The first two indicate that the tags in the grammar follow Semantic Interpretation (SI) script format while the last two represent SI string literals format. Other tag formats are not supported at this point. If the tag format declaration is missing from the grammar header, SI string literals format will be used.

Note that language and mode declarations are optional because VASRE supports only one language and one mode at this point. That is, the following two grammars are identical.

```
#ABNF 1.0;
language en-us;
mode voice;
root $yes;

$yes = yes [please];
```

```
#ABNF 1.0;
root $yes;

$yes = yes [please];
```

Other standard SRGS declarations such as base URI, pronunciation lexicon, and tag are not supported at this point.

3 Rule Definition

A rule definition associates the rule name with the rule description, which represents what sequences of words, sub-rules, and tags can be matched to a user utterance. The rule definition comprises the rule name, assignment operator (=), rule description, and a semicolon. The rule name may follow the scope keyword *public* or *private*. However, this is meaningless since the current version of VASRE does not support external rule reference.

The rule used as a sub-rule of a certain rule must be defined somewhere within the same grammar file. For example, the following example grammar is wrong because the definition of *$animal* is missing:

```
#ABNF 1.0;
root $root;
$root = $animal;
```

The recursive definition of rule(s) is not supported. For example, the following two grammars are syntactically wrong:

```
#ABNF 1.0;
root $root;
$root = $root one; // error
```

```
#ABNF 1.0;
root $root;
$root = $a;
$a = $b two;
$b = three $a; // error
```

4 Rule Names and Tokens

The rule name must start with a dollar-sign character *$*. The string following the dollar sign must be a combination of alphabetical and numerical characters and underscores starting with an alphabetical letter. For example, *$animal* and *$rule_1* are valid rule names, but *$1st* and *$rule%2* are not. The maximum allowed length of a rule name is 64.

A token is the part of a text grammar that defines one or more words that can be matched. All the fields other than rule names, tags, and operators in the rule definition part delimited by white spaces are tokens. Alphabetical letters, apostrophes, quotation marks, hyphens, underscores, and periods can be used to compose a token string, but a numerical character cannot. A token doesn't need to start with an alphabetical letter, but must contain at least one alphabetical letter. Also, the adjacency of the following letters is not allowed: apostrophes, quotation marks, hyphens, and underscores.

Normally, a token comprises a single word. However, you can combine multiple words to build a token using quotation marks, hyphens, underscores, or periods. For example, *San Francisco* represents two tokens delimited by the white space while *"San Francisco"*, *San-Francisco*, *San_Francisco*, and *San.Francisco* represent a single token.

N.B. Note that quotation marks, hyphens, underscores, and periods never affect the pronunciations of the token. For example, *a-b* and *company.com* are not pronounced **a dash b** and **company dot com** but pronounced **a b** and **company com**, respectively.

Merging multiple words into a single token has two advantages. First, a token is regarded as the minimum unit of the grammar during speech recognition stage and you will have a single confidence score for each token. For example, consider the following two grammars representing the alternatives of four names (See Section 8 for further details on alternative operators):

```
#ABNF 1.0;
root $name;
$name = madison smith
      | alex johnson
      | tyler williams
      | landon jones
      ;
```

```
#ABNF 1.0;
root $name;
$name = "madison smith"
      | alex-johnson
      | tyler_williams
      | landon.jones
      ;
```

If **madison smith** was recognized for the grammar on the left-hand side, two confidence scores will be output, one for each of **madison** and **smith**. On the other hand, the grammar on the right-hand side will output a single confidence score for all of the four names listed. This is useful when you have a long list of short keywords to be recognized and want to use the corresponding single confidence score in your application.

The second advantage is that if the grammar represents the alternatives of short keywords, making each keyword a single token yields better recognition results than just enumerating them. Recall the above two example grammars and assume that they enumerate 100 names instead of four. Then, the right-hand side grammar will have slightly better recognition performance than the left-hand side one, particularly when the grammars are tested with non-native speakers. Therefore, it is highly recommended to represent the grammar as alternatives of single tokens if you will expect your system to recognize one amongst many short keywords.

The word-merging characters (except double quotation marks) appear in the recognition results. For instance, recall the above example grammar on the right-hand side. All the possible recognition results

from the grammar are *madison smith*, *alex-johnson*, *tyler_williams*, and *landon.jones*. Note also that a token delimited by quotation marks takes a white space normalization step, which replaces successive white spaces and tabs with a single white space and strips off white spaces at the beginning and end. For example, all of the following tokens will appear as *madison smith* in the recognition outputs:

"madison smith", *"madison smith"*, *" madison smith "*.

In some situations, it is onerous to list all the possible phrases to be recognized as single tokens. For example, suppose that we were to write a grammar recognizing two-digit numbers. Using the repetition operator that will be introduced in Section 11, the grammar can be written simply as follows:

```
#ABNF 1.0;
root $root;
$root = $digit<2>;
$digit = one|two|three|four|five|six|seven|eight|nine|zero;
```

To improve the recognition performance, you may attempt to enumerate all of the 100 possible phrases as single tokens. But, this task will be very tedious and mistakes could easily be made. Instead, you could write the grammar using repetition operators and make the grammar compiler enumerate all the possible phrases represented by the grammar as single tokens using the `-kwd` option. See Section 1.4 or 2.4 of Administration Guide for further details.

N.B. If the `-kwd` option is used when compiling the grammar, weights (See Section 8) and tags (See Section 13) in the text grammar will be ignored.

Apostrophes are another special character that can be used for token strings. Note that apostrophes are used to represent some common English words, not to merge multiple words into a single token. For example, the pronunciation of *can't* will be correctly generated as *k ae n t* while the pronunciation of *can_t* will be *k ae n t iy.*, which sounds like candy. This is because *can_t* represents a merged token of *can* and *t*.

Other characters, such as numerical, cannot be used for token strings. For example, 1, 22, dog@home, and hundred% are unacceptable tokens. Use one, twenty two, dog at home, and hundred percent instead. Note also that the length of a token is limited to 64.

5 Blank Spaces and Cases

The user may use as many blank spaces, new lines, or tabs between two fields in rule definitions as desired. For example, the following two grammars are identical:

```
#ABNF 1.0;
root $root;
$root = one
      | two
      | three;
```

```
#ABNF 1.0;
root

$root;
$root = one | two   | three;
```

Note that a new line will be regarded as a white space while processing the rule definition. For example, the rule *$root* in the following example represents a token of two words *by* and *pass*, not a single word *bypass*.

```
#ABNF 1.0;
ROOT $root;
$root = by
pass;
```

The grammar text is case-insensitive except for rule names. For example, the following two grammars are equivalent to each other:

```
#ABNF 1.0;
root $root;
$root = one|two|three;
```

```
#ABNF 1.0;
ROOT $root;
$root = oNe|TWo|thrEE;
```

Tokens are converted into lower case internally. For the second grammar above, if **one** is recognized, you will have *one* as the recognition output instead of *oNe*.

A rule name is case-sensitive. For example, *$rule*, *$Rule*, and *$RULE* are different from one another and the following grammar is syntactically wrong:

```
#ABNF 1.0;
root $root;
$ROOT = one|two|three;
```

6 Comments

Both C and C++ style comments are allowed in the text grammar as illustrated below:

```
#ABNF 1.0;
/*
 * This grammar is for digit recognition
 */
ROOT $root;
$root = zero // "oh" is not allowed
  |one|two|three|four|five|six|seven|eight|nine;
```

The commented sections will be ignored during grammar compilation.

7 Phonetic Spelling

Within the text grammar, you can specify pronunciations of a user-defined token. This convention called phonetic spelling can be used when you wish to define the pronunciations of a certain token without relying on the auto pronunciation feature of `GramGen`. See Section 1.4 or 2.4 of Administration Guide for further details.

The phoneme symbols listed below can be used for US-English phonetic spelling:

Phoneme	Example	Transcription	Phoneme	Example	Transcription	Phoneme	Example	Transcription
aa	car	k aa r	f	father	f aa dh er	p	pop	p aa p
ae	bat	b ae t	g	gag	g ae g	r	rerun	r iy r ah n
ah	butter	b ah t er	hh	hockey	hh aa k iy	s	score	s k ao r
ao	ought	ao t	ih	it	ih t	sh	share	sh eh r
aw	count	k aw n t	iy	clean	k l iy n	t	total	t ow t ah l
ay	bite	b ay t	jh	judge	jh ah jh	th	theme	th iy m
b	bob	b aa b	k	kick	k ih k	uh	book	b uh k
ch	church	ch er ch	l	lily	l ih l iy	uw	two	t uw
d	dad	d ae d	m	mom	m aa m	v	very	v eh r iy
dh	they	dh ey	n	none	n ah n	w	we	w iy
eh	bet	b eh t	ng	sing	s ih ng	y	yet	y eh t
er	bird	b er d	ow	boat	b ow t	z	zoo	z uw
ey	sale	s ey l	oy	boy	b oy	zh	measure	m eh zh er

Besides 39 phoneme symbols listed above, you may use *sil*, which represents silence. Inserting *sil* will be helpful particularly when you expect a short pause or silence within the recognized speech.

A string embraced within the pair of quotation marks and curly brackets describes the pronunciation and the corresponding token. The string starts with a list of phonemes separated by white spaces followed by a colon and the token string. Consider the following example:

```
#ABNF 1.0;
root $company;
$company = "{v eh s t eh k:vestec}";
```

If a speech represented by *v eh s t eh k* is recognized with this grammar, *vestec* will appear as the corresponding recognition output.

Several basic rules of phonetic spellings are as follows:

1. No white spaces are allowed between the quotation mark and curly bracket.
2. The token must be specified. That is, you may not specify the pronunciation without specifying the matched token.

For example, the following two grammars are syntactically wrong:

```
#ABNF 1.0;
root $root;
$root = " {v eh s t eh k:vestec}"; // error
```

```
#ABNF 1.0;
root $root;
$root = "{v eh s t eh k}"; // error
```

The token string representing a single token may comprise multiple words. You may use white spaces, double quotation marks, hyphens, underscores, and periods for the token string. Regardless how many words are used for the token name, they will be regarded as a single token. For example, the following two grammars identically represent a single token *"a vestec stock"*:

```
#ABNF 1.0;
root $root;
$root = "{ah v eh s t eh k s t aa k:a vestec stock}";
```

```
#ABNF 1.0;
root $root;
$root = "{ah v eh s t eh k s t aa k:"a vestec stock"}";
```

The strings representing phoneme sequence and token name go through white space normalization. For example, the following two grammars are identical:

```
#ABNF 1.0;
root $root;
$root = "{ah v eh s t eh k s t aa k:a vestec stock}";
```

```
#ABNF 1.0;
root $root;
$root = "{ah   v eh   s t eh k s t aa k
         :    a     vestec   stock  }";
```

Never attempt to use phonetic spelling as a part of a token comprising multiple words because it might be confusing. The grammar will compile, but in a way different from what you intended. Consider the following example:

```
#ABNF 1.0;
root $root;
$root = "a "{v eh s t eh k:vestec}" stock";
```

With this grammar, the author intended to recognize a token pronounced **a vestec stock**. But, the grammar compiler understands this grammar as the sequence of two tokens *"a "* and *" stock"*, where *v eh s t eh k:vestec* is the tag statement following *"a "*. See Section 13 for further details. Likewise, the following grammar can be understood in two different ways: a single token saying **vestec com** or double tokens of *vestec* and *.com*. The grammar compiler follows the latter way.

```
#ABNF 1.0;
root $root;
$root = "{v eh s t eh k:vestec}".com;
```

If you want to define multiple pronunciations of a user-defined token, you can separate them by commas as illustrated in the following example:

```
#ABNF 1.0;
root $Name;
$Name = "{k ae r ah l ay n, k ae r ah l ih n:caroline}" miller;
```

To see if a certain word is contained in the pronunciation dictionary, use the pronunciation generator. If you compile a grammar containing a word whose pronunciation is not defined, the grammar compiler will output an error message. However, you may use the `-ap` option to make the grammar compiler guess the pronunciation. See Section 1.4 or 2.4 of the Administration Guide for further details.

It is not recommended to redefine the pronunciation of a word that already exists in the dictionary. If you do this, the original pronunciation will be ignored within the scope of that text grammar. Consider the following example:

```
#ABNF 1.0;
root $ROOT;
$ROOT = "{y eh s sil p l iy z:yes}";
```

This grammar maps the phoneme sequence *y eh s sil p l iy z* to *yes*. The original pronunciation of *yes* will be ignored by this grammar. However, if *yes* appears somewhere in the grammar as in the following example, both *y eh s* and *y eh s sil p l iy z* will be used as pronunciations of *yes*:

```
#ABNF 1.0;
root $ROOT;
$ROOT = $yes|$yesplz;
$yes = yes;
$yesplz = "{y eh s sil p l iy z:yes}";
```

The following grammar is even more problematic. It will represent **yes yes**, which can be pronounced in four different ways: *ae ae*, *ae y eh s*, *y eh s ae*, *y eh s y eh s*.

```
#ABNF 1.0;
root $ROOT;
$ROOT = yes "{ae:yes}";
```

8 Alternatives and Weights

The basic operator used for rule description is an alternative operator, which expands a rule into two or more sequences of words or sub-rules. A vertical bar is used as the alternative operator. For example, the following grammar is supposed to recognize **one**, **two**, or **three**.

```
#ABNF 1.0;
root $root;
$root = one|two|three;
```

An empty alternative is not allowed. That is, the following three grammars are syntactically wrong:

```
#ABNF 1.0;
root $root;
$root = |two|three; // error
```

```
#ABNF 1.0;
root $root;
$root = one||three; // error
```

A weight can be optionally paired with an alternative. A weight is a simple positive floating point value without an exponential symbol. An alternative with the weight of 1.0 has the same effect to an alternative with no weight. A weight greater than 1.0 positively biases the alternative while a weight less than 1.0 negatively biases the alternative.

Use a pair of slashes to specify the weight of an alternative as shown in the following example:

```
#ABNF 1.0;
root $root;
$root = /2.0/ coffee | /1./ tea | /0.5/ $others;
$others = cookie | /.5/ donut;
```

A weight can be assigned to a sub-rule. If this is the case, the weight is multiplied by the alternatives the sub-rule defines. That is, the above example grammar is equivalent to the following:

```
#ABNF 1.0;
root $root;
$root = /2.0/ coffee | /1./ tea | /0.5/ cookie | /.25/ donut;
```

Note that you should use a reasonable range for the weight. If the weight is greater than 1000 or less than 0.001, it will be set to 1000 or 0.001, respectively, by the grammar compiler. That is, the weight of 1000000000 has the same effect of the weight of 1000.

N.B. If the `-kwd` option is used when compiling the grammar, weights and tags (See Section 13) in the text grammar will be ignored.

9 Priority Parentheses

You may specify priority in rule description using parentheses. For example, the following two grammars are identical:

```
#ABNF 1.0;
root $root;
$root = give me (bills | coins) please;
```

```
#ABNF 1.0;
root $root;
$root = give me bills please
          | give me coins please;
```

Priority parentheses embracing an empty string will be ignored.

10 Option Brackets

You may specify optional words or sub-rules in a rule description using square brackets. For example, the following two grammars are identical:

```
#ABNF 1.0;
root $root;
$root = yes [please];
```

```
#ABNF 1.0;
root $root;
$root = yes | yes please;
```

Option brackets embracing an empty string are not allowed.

11 Repetition Brackets

You may specify repetition in rule description using angle brackets. Write the number of repetitions, or the minimum and maximum numbers of repetitions separated by a hyphen within the brackets. For example, the following two grammars are identical:

```
#ABNF 1.0;
root $root;
$root = well <0-2> umm <2>;
```

```
#ABNF 1.0;
root $root;
$root = umm umm
      | well umm umm
      | well well umm umm;
```

The minimum number of repetitions must be no less than zero, and there exists no limitation on the maximum repetition number. If your grammar allows infinite repetition of a certain rule or terminal, specify the minimum repetition number only using a hyphen as shown in the following example:

```
#ABNF 1.0;
root $root;
$root = wow <1->;
```

However, omitting the minimum repetition number is not allowed:

```
#ABNF 1.0;
root $root;
$root = wow <-10>; // error
```

If the repetition brackets follow priority parentheses, the grammar described within the priority parentheses are repeated. For example, the following grammar represents 4 to 6-digit numbers:

```
#ABNF 1.0;
$root = (zero|one|two|three|four|five|six|seven|eight|nine) <4-6>;
```

You may give a weight to repetition. If the weight is given in front of the repeated words or sub-rules, the weight will be identically applied to all the phrases represented by the repetition. For example, the following grammar is biased towards **wow** and **wow wow** with the weight of two.

```
#ABNF 1.0;
root $root;
$root = /2./ wow <1-2> | oh;
```

Note that giving weight to the words or sub-rules repeated has no effect if there are no alternatives. The following two grammars are equivalent from the server's point of view:

```
#ABNF 1.0;
root $root;
$root = (/.7/ wow) <1-2>;
```

```
#ABNF 1.0;
root $root;
$root = (/2/ wow) <1-2>;
```

If the weight is given next to the number of repetitions, it would indicate the probability of successive repetition of the words or sub-rules. For this case, weight must be in the range of 0.0 to 1.0. For example, the following grammar indicates that the chance of matching **wow wow** is 70% and the chance of **wow wow wow** is 49%.

```
#ABNF 1.0;
root $root;
$root = wow <1-3 /.7/>;
```

If maximum number of repetition is not specified, the probabilities decay exponentially.

As a summary, consider the following grammar:

```
#ABNF 1.0;
root $root;
$root = /2./ wow <1- /.5/>) | oh;
```

This grammar is positively biased towards the repetition of *wow* with the weight of two and the chance to match additional *wow* will be decayed with the probability of 0.5.

12 Special Rules

Three rule names *$NULL*, *$VOID*, and *$GARBAGE* are reserved for special recognition purposes. These rules are so-called special rules, and must not be redefined within a grammar.

$NULL defines a rule that is automatically matched even when nothing was spoken. For example, the following two grammars are identical:

```
#ABNF 1.0;
root $root;
$root = yes $NULL please;
```

```
#ABNF 1.0;
root $root;
$root = yes please;
```

Note that the root rule must not represent *$NULL*. For example, the following grammar is syntactically wrong.

```
#ABNF 1.0;
root $root;
$root = $NULL;
```

$VOID defines a rule that can never be matched. For example the following two grammars are identical:

```
#ABNF 1.0;
root $root;
$root = apple | $VOID;
```

```
#ABNF 1.0;
root $root;
$root = apple;
```

The root rule must not be *$VOID*.

$GARBAGE defines a rule that may match any speech up until the next sub-rule or word matches. For example, the following grammar covers every speech starting with yes including yes itself.

```
#ABNF 1.0;
root $root;
$root = yes $GARBAGE;
```

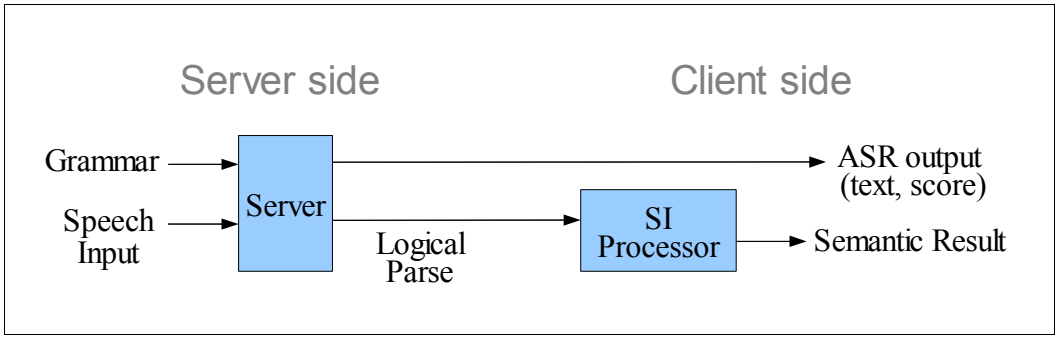
Note that neither *$NULL* nor *$GARBAGE* will ever appear as a recognition result, even when they match a part of input speech. When the grammar compiler generates the sentences represented by the grammar containing special rules, *$NULL* and *$VOID* will not appear as a part of the sentences, but *$GARBAGE* will. See Section 2.4 or 3.4 of the Administration Guide for further details.

It is not recommended to use *$GARBAGE* without extensive testing. The capability of *$GARBAGE* is limited and it may degrade the overall recognition performance of your speech recognition grammar.

13 Tags

Within a text grammar, you may leave tag statements within curly brackets so that the SI processor can use it to generate a semantic result. The tag statement surrounded by curly brackets is called a tag. Following SISR Version 1.0, the tag statement can be written in two formats: SI script and SI string literals. SI script is a source code of European Computer Manufacturers Association (ECMA) program assigning the result of evaluation to the rule variable that the tag is subject to. SI string literals specify a string comprising one or more characters that will be assigned to the rule variable. The choice of tag format is specified by tag declaration in the header. See Section 2 for further details.

Based on tags in the rule expansion, the recognition server generates a logical parsing, which describes the hierarchy of matched rules and the words and tags therein. The logical parsing contains all the required information for generating semantic result. The server outputs the logical parsing as part of the recognition result so that the client program can invoke the SI processor with it. The data flow between the recognition server and SI processor is illustrated below. See Section 2.3, 2.4, and 6 of Application Developer's Guide for further details.



N.B. If the `-kwd` option is used when compiling the grammar, weights (See Section 8) and tags in the text grammar will be ignored. In this section, we assume all the example grammars were compiled without `-kwd` option.

See the following example:

```
#ABNF 1.0;
tag-format <semantics/1.0-literals>;
root $yesno;
$yesno = yes {TRUE} | no {FALSE};
```

Since *tag-format <semantics/1.0-literals>;* is declared, this grammar follows SI string literals format. If **yes** is recognized, the logical parsing will be

[$yesno[yes, {TRUE}]]

This logical parsing assigns the string *TRUE* to the rule variable of *$yesno*, which is the root rule of this grammar. So, *TRUE* will be returned as the corresponding semantic result. The equivalent grammar with SI script format can be written as follows:

```
#ABNF 1.0;
tag-format <semantics/1.0>;
root $yesno;
$yesno = yes {out = "TRUE";} | no {out = "FALSE";};
```

The logical parsing from this grammar will be:

[$yesno[yes, {out = "TRUE";}]]

The script *out = "TRUE";* assigns the string literal *"TRUE"* to the rule variable *$yesno*.

VASRE version 1.1 fully supports SISR version 1.0. See SISR document available on W3C website for full descriptions on SISR. The rest of this section summarizes the key points of SISR with some examples.

13.1 Global Tags

As a part of the grammar header, global tags can be declared. The global tags will be added to the beginning of the logical parsing, so that it could be used while evaluating the tag statements in rule expansions.

Consider the following example:

```
#ABNF 1.0;
tag-format <semantics/1.0>;
root $yesno;
{var t = "TRUE"; f = "FALSE";};
$yesno = yes {out = t;} | no {out = f;};
```

If **yes** is matched, the logical parsing will be:

[{var t = "TRUE"; f = "FALSE";}, $yesno[yes, {out = t;}]]

This logical parsing will output the semantic result of *"TRUE"*.

13.2 Rule Variables

Three variables are mapped to every rule in the grammar:

- Rule variable named *out*
- Text variable named *text*
- Score variable named *score*

The rule variable *out* holds semantic value of the rule. The type of the rule variable can be one of string, integer, double, boolean, object, and array. You don't need to declare the type of the rule variable. It will be automatically decided based on the value assigned to the rule variable.

Rule variable may have one or more properties (child variables). The property can be individually accessed by *out.identifier*, where *identifier* is the name of the property. For example, *out.pizza* represents the pizza property of the rule variable. Reserved words of ECMAScript such as *for* must not be used as property names. If a rule variable has properties, its type will be set to object.

text of a rule holds the sub-string in the utterance that is governed by the corresponding rule. *score* holds a value that is related to the confidence probability of the corresponding rule. Note that both *text* and *score* are read-only from SI tag's point of view.

The rule variable of the root rule is the semantic result of a grammar.

13.3 Syntax for Variables

Every SI script in tags has access to an object variable named *rules* that has a property holding the rule variables value of rules. The rule variable associated to a rule reference is identified by *rules.rulename*, where *rulename* is the name of the rule. Individual properties of a rule variable can be identified by *rules.rulename.identifier*, where identifier is the name of the property.

The rule variable for the latest rule reference that was used in the expansion matching the utterance up to the position of the SI tag can also be referenced through *rules.latest()*. Rule variables of the current rule and referenced rules can be evaluated and assigned to. Special rules such as *$NULL*, *$VOID*, and *$GARBAGE* cannot be evaluated.

The access to the text and score variables of rules are available in the form of the followings:

- *meta.rulename.text*
- *meta.latest().text*
- *meta.current().text*
- *meta.rulename.score*
- *meta.latest().score*
- *meta.current().score*

Note again that these variables are all read-only.

For example, consider the following example:

```
#ABNF 1.0;
language en-US;
tag-format <semantics/1.0>;
root $drink;
$drink =
       {rules.size="medium"} $size<0-1> $kind
       {out.drinksize=rules.size; out.type=rules.kind;}
       ;
$size = small | medium | large;
$kind = coke | pepsi;
```

The rule *drink* has a read and write access to the rule variables of referenced rules *size* and *kind*. The SI tags declare two properties for the rule variable of *drink*: *drinksize* and *type*, which are set to the rule variable values of *size* and *kind*.

13.4 Default Assignment

If no SI tag is attached to the expansion of a certain rule that is used to match the utterance, then the value of the rule variable *out* for the rule is determined as follows. If there are no rule references in the parsing, the value of *meta.current().text* is automatically copied into the rule variable (which then becomes of type string). Otherwise, the value of the rule variable of the last rule reference in the parsing (which is named *rules.latest()*) is automatically copied into the rule variable.

For example, the value of the rule variable *$drink* in the following example is either *coke*, *pepsi*, or *coca cola*.

```
$drink = coke | pepsi | coca cola;
```

For the following example, however, *rules.drink* is either *coke* or *pepsi* while *meta.drink.text* is one amongst *coke*, *pepsi*, or *coca cola*.

```
$drink = coke | pepsi | coca cola {coke};
```

14 Vocabulary Size

The vocabulary size of the VASRE is limited to 500, which means the total vocabulary size of the grammars added to the VASRE grammar list may not exceed 500. Therefore, you need to know how vocabulary size is counted for a given text grammar to maintain total vocabulary size below the limit.

The vocabulary size of a grammar is the summation of vocabulary sizes of rules defined in the grammar. The vocabulary size of a rule is the number of word segments separated by sub-rules, alternative operators, parentheses, and brackets. Note that the vocabulary size has nothing to do with the actual number of the words appearing in the recognition result.

See the comments in the following sample grammar to be familiar with the counting method described above.

```
#ABNF 1.0;
root $vs;
$vs = $vs1 | $vs2 | $vs3 | hello there;   // vocabulary size: 1
$vs1 = yes [please] | no [thanks];        // vocabulary size: 4
$vs2 = "{g uw g l:google}" please;        // vocabulary size: 1
$vs3 = give me (bills | coins) please;    // vocabulary size: 4
```

The total vocabulary size of the above sample grammar is 10.

15 Sample Grammars

The VASRE SE comes with eight sample grammars under `/opt/VestecASRE/Samples/Grammars/` for GNU/Linux and `VestecASRE\Samples\Grammars\` for Windows:

1. `Date.grm` matches a date in Month-Day-Year format. An example of matched utterance is **july the third two thousand eight**. This grammar declares the tag format of <*semantics/1.0*> and contains SI tags to output a semantic result with three properties: year, month, and day.
2. `Digit4to6.grm` matches 4- to 6-digit number such as **one five six zero**, **four three three zero one**, and **five six eight seven eight nine**. This grammar contains SI tags to output a semantic result of type string representing 4- to 6-digit numbers.
3. `Digit5.grm` matches 5-digit numbers: **three six four nine one**. This grammar contains SI tags to output a semantic result of type string representing 5-digit numbers.
4. `Money.grm` matches an amount of money from 1 cent to 100 dollars and 99 cents: **five dollars and fifty cents**, **hundred dollars**, **six cents**. This grammar contains SI tags to output a semantic result of type double representing the dollar amount.
5. `Name.grm` matches 100 American names such as **alex johnson**, **jackson clark**.
6. `Number.grm` matches a number between 0 and 999,999: **forty six**, **fifty two sixty three**. This grammar contains SI tags to output a semantic result of type integer.
7. `Time.grm` matches time of day: **twelve o'clock in the morning**, **half past twelve**, **three a m**.
8. `Yesno.grm` matches short phrases representing yes or no: **yes please**, **sure**, **no thanks**, **absolutely not**. This grammar declares the tag format of <*semantics/1.0-literals*> and contains SI tags to output a semantic result of type string, which can be either *TRUE* or *FALSE*.

For further details on each grammar, refer to the comments in `.grm` files.

Appendix: List of Text Grammar Error Codes

The grammar compiler of VASRE outputs an error code if the given grammar has syntax errors. For example, if the self-identifying header is missing from the text grammar, `GramGen` will output the following message:

```
Loading pronunciation dictionary ...
--- Done.
Preprocessing .grm file ...
--- The grammar must start with self-identifying header "#ABNF 1.0;". (Code: 55)
```

For this example, 55 is the error code. The error message followed by the error code gives a very basic description of the error. To obtain more details on the error, refer to this section. The error codes are in a decimal format and arranged in an ascending order.

N.B. If the error message starts with *Internal Error* and a colon, the error happened due to defects in the grammar compiler itself. Please report it with the error code to Vestec if you encounter internal errors.

30: Failed to open .grm file

The grammar compiler could not open the `.grm` file. Ensure that the `.grm` file exists and its permission is correctly specified.

31: .grm file must be a text file

The grammar compiler detected that the `.grm` file is not an ascii text file. Check whether the `.grm` file is a valid ascii text file containing SRGS text. Character encoding other than US-ASCII is not supported.

40: Missing semicolon

A semicolon is missing in the last line of the text grammar.

41: Running comment

Comment is not closed. Check the pair of /* and */. See Section 6 for details.

42: Line is too long

A certain line in the text grammar is too long. The maximum allowed line length is 1022.

43: Running tag

Tag is not closed. Check the pair of *{* and *}*.

44: Running phonetic spelling

Phonetic spelling is not closed. Check the pair of *"{* and *}"*.

45: Running token

Token is not closed. Check the pair of *"* and *"*.

50: Empty grammar file

The grammar file is empty.

51: Vocabulary size overflows

The vocabulary size of the text grammar exceeds the limit 500. See Section 14 to learn how to count the vocabulary size of a grammar.

52: Special rules as root rule

Special rules *$NULL*, *$VOID*, and *$GARBAGE* must not be declared as the root rule. See Section 12.

53: Root grammar represents *$NULL*

The grammar represents *$NULL*. A grammar representing *$NULL* is not allowed for the VASRE grammar compiler. See Section 12 for details.

55: Missing self-identifying header

The grammar must start with the self-identifying header starting with *#ABNF*. See Section 2 for further details.

56: SRGS version is missing

No SRGS version number is found. The self-identifying header keyword *#ABNF* must be followed by the SRGS version number *1.0*. See Section 2 for further details.

57: SRGS version is not supported

The SRGS version number following *#ABNF* is not supported by the VASRE grammar compiler. Only version 1.0 is supported and the self-identifying header must be *#ABNF 1.0;*. See Section 2 for further details.

58: Character encoding is not supported

The character encoding specified in the self-identifying header is not supported. Only *US-ASCII* is supported. See Section 2 for further details. This error can be also issued when the semicolon is missing at the end of the self-identifying header.

59: Unknown header keyword

The grammar compiler encountered unknown keywords while processing the grammar header.

61: Phonetic spelling delimiters mismatch

The grammar compiler found a mismatch of phonetic spelling delimiters comprising double quotation marks and curly braces. Check the pair of the delimiters.

62: Invalid root rule name

Invalid root rule name follows the keyword *root*. You may encounter this error because a semicolon is missing at the end of the root rule declaration.

63: Invalid language identifier

Invalid language identifier follows the keyword *language*. Only *en* or *en-us* can be used as language identifiers. You may encounter this error because a semicolon is missing at the end of the language declaration.

64: Invalid mode identifier

Invalid mode identifier follows the keyword *mode*. Only *voice* can be used as a mode identifier. You may encounter this error because a semicolon is missing at the end of the mode declaration.

65: Double quotation marks mismatch

The grammar compiler found a mismatch of double quotation marks representing a token comprising multiple words. Check the pair of the double quotation marks.

66: Base URI is not supported

Base URI is declared, but the VASRE grammar compiler does not support it. See Section 2 for details.

67: Pronunciation lexicon is not supported

Pronunciation lexicon is declared, but the VASRE grammar compiler does not support it. See Section 2 for details.

68: Meta data is not supported

Meta data is declared, but the VASRE grammar compiler does not support it. See Section 2 for details.

69: Weight is not supported

Weights of tokens or sub-rules are not supported within rule definition.

70: Root rule declaration in grammar body

Root rule declaration is found after one or more rules are defined. The root rule must be declared before rules are defined.

71: Missing rule description

A rule definition must comprise a rule name, an assignment operator, and the rule description. The rule description is missing. See Section 3 for details.

72: Rule redefined

The grammar compiler found multiple definitions of a certain rule. A rule must be defined only once.

73: Missing assignment operator

A rule definition must comprise a rule name, an assignment operator, and the rule description. The assignment operator is missing. See Section 3 for details.

74: Language identifier is not supported

A language identifier is not supported for the current version of VASRE grammar compiler.

75: No white space follows *root*

The root rule declaration must comprise the keyword *root*, a white space, and the root rule name, but no white space character is found. See Section 2 for details.

76: No white space follows *language*

The language declaration must comprise the keyword *language*, a white space, and the language identifier, but no white space character is found. See Section 2 for details.

77: No white space follows *mode*

The mode declaration must comprise the keyword *mode*, a white space, and the mode identifier, but no white space character is found. See Section 2 for details.

78: No white space follows *tag-format*

The tag format declaration must comprise the keyword *tag-format*, a white space, and the tag format identifier, but no white space character is found. See Section 2 for details.

79: Invalid tag format identifier

Invalid tag format identifier follows the keyword *tag-format*. Only <*semantics/1.0-literals*>, <*semantics/1.0.2006-literals*>, <*semantics/1.0-literals*>, and <*semantics/1.0.2006-literals*> can be used as tag format identifier. You may encounter this error because a semicolon is missing at the end of tag format declaration. See Section 2 for details.

80: No white space follows *public*

No white space is found between the keyword *public* and rule name.

81: No white space follows *private*

No white space is found between the keyword *private* and rule name.

82: Duplicate root rule declaration

The grammar compiler found multiple declarations of the root rule. See Section 2 for details.

83: Duplicate language declaration

The grammar compiler found multiple declarations of the language. See Section 2 for details.

84: Duplicate mode declaration

The grammar compiler found multiple declarations of the mode. See Section 2 for details.

85: Duplicate tag-format declaration

The grammar compiler found multiple declarations of the tag format. See Section 2 for details.

86: Special rules are defined

Special rules *$NULL*, *$VOID*, and *$GARBAGE* are reserved and cannot be defined.

87: No matching tag opening brace

The grammar compiler encountered *}!}*, but there exists no matching *{!{*.

88: No token

Double quotation marks surround an empty string.

90: Rule description starting with alternative operator

The grammar compiler found a rule description starting with an alternative operator. The alternative must follow tokens or sub-rules.

91: Rule description ending with alternative operator

The grammar compiler found a rule description ending with an alternative operator. The alternative must be followed by tokens or sub-rules.

92: Subsequent alternative operators

The grammar compiler found two or more subsequent alternative operators. Tokens or sub-rules must be inserted between the alternative operators.

94: Missing tag format

Tag format identifier is missing after *tag-format* keyword. Use one of the following identifiers, <*semantics/1.0*>, <*semantics/1.0.2006*>, <*semantics/1.0-literals*> or <*semantics/1.0.2006-literals*>.

95: Invalid global tag

The global tag contains an empty statement or closing tag brace is not found.

100: Invalid phoneme in phonetic spelling

A phoneme symbol used for phonetic spelling is invalid. See Section 7 for the full list of phonemes.

101: Missing token in phonetic spelling

A phonetic spelling must comprise phoneme sequences, a colon, and a token string. The token string is missing from the phonetic spelling.

102: Missing pronunciations in phonetic spelling

A phonetic spelling must comprise phoneme sequences, a colon, and a token string. The phoneme sequences are missing from the phonetic spelling.

112: Invalid Token

The grammar compiler expected a token string comprising alphabetical letters, hyphens, underscores, and periods, but encountered something else. Note that numerical letters and special characters other than hyphens, underscores, and periods cannot be used as a part of a token string.

113: Token is too long

A token is too long. The maximum allowed length of a token string is 64.

114: Invalid Token

A token contains no alphabetical letter(s). It must contain at least one alphabetical letter.

115: Invalid Token

A token contains subsequent apostrophes, hyphens, underscores, or periods. An apostrophe, hyphen, underscore, or period must not be followed by another apostrophes, hyphen, underscore, or period.

116: Invalid Token

The token starts with a double quotation mark, but ends with a character other than the double quotation mark.

130: Invalid rule name

Rule name is too short.

131: Invalid rule name

Rule name must start with a dollar sign.

132: Invalid rule name

Rule name must start with a dollar sign followed by an alphabetical character.

133: Invalid rule name

Rule name must start with a dollar sign followed by an alphabetical character. The rest must comprise alphabetical and numerical characters and underscores. Other character cannot be used as a part of rule names. You may encounter this error if you miss an assignment from a rule definition. See Section 4 for further details.

134: Invalid rule name

Rule reference is found in the grammar, but the VASRE grammar compiler does not support it.

135: Invalid rule name

Rule name is too long. The length of a rule name must not exceed 64.

140: Missing root rule definition

The definition of the root rule is missing.

141: Missing rule definition

A rule is used, but its definition is missing.

200: Initialization failed

The grammar compiler failed to load pronunciation dictionary. Ensure `agg_en1.bin` is under `/opt/VestecASRE/bin/`.

250: Pronunciation not found

The pronunciation of the token is not found in the dictionary.

251: Auto pronunciation failed

Auto pronunciation failed to guess the pronunciation of the token.

252: Token has too many pronunciations

The token has more than 100 pronunciations. This may occur when you build a token by concatenating multiple words and each word has multiple pronunciations. Divide the long token into several pieces to resolve this error.

271: Too long logical parsing

While the grammar compiler generates the segments of logical parsing, it encountered a too long segment, which cannot be placed within the binary grammar file. This may happen if too long tags exist in the text grammar.

480: Weight slash mismatch

The grammar compiler found a mismatch of weight slashes. Check the pair of weight slashes.

481, 485: Missing weight

The grammar compiler found two successive slashes. Place the weight between them.

482, 483: Invalid weight

The format of the weight is wrong. Legal formats are *n*, *n.*, *.n*, and *n.n*, where *n* is a sequence of one or more digits.

484: Invalid weight

The grammar compiler found a wrong weight. The weight must be a simple positive number.

486: Weight slash mismatch within repetition brackets

The grammar compiler couldn't find the second slash within a repetition bracket. Check the pair of slashes for weight.

487: Invalid weight

The weight for repetition must be in the range of 0 to 1. Negative weight or positive weight more than 1 is not allowed.

490: Priority parentheses mismatch

The grammar compiler found a mismatch of priority parentheses. Check the pair of priority parentheses.

491: Option brackets mismatch

The grammar compiler found a mismatch of option brackets. Check the pair of option brackets.

492: Repetition brackets mismatch

The grammar compiler found a mismatch of repetition brackets. Check the pair of repetition brackets.

493: Parentheses or brackets mismatch

The grammar compiler found a mismatch of priority parentheses or option brackets. Check the pair of priority parentheses and option brackets.

495: Tag braces mismatch

The grammar compiler found a mismatch of tag braces. Check the pair of interpretation brackets.

496: No string in tag

Tag braces embrace no string.

497: Wrong place of weight

Weight must precedes a token or a rule to be weighted.

498: Tag statement is too long

Tag statement is too long. The maximum allowed length of a tag statement is 1024.

499: Compiling *$GARBAGE* with -kwd

If *$GARBAGE* is used, the grammar cannot be compiled with the `-kwd` option. Remove the `-kwd` option to resolve this error.

500: No string within option brackets

Option brackets embrace no string.

501: Tag statement contains closing bracket

The grammar compiler found a tag statement containing a closing bracket *}*. You may not use the closing bracket as a part of the tag statement.

502: Recursive rule definition

The grammar compiler found recursive definition of rule(s), which is not supported by the VASRE grammar. See Section 3 for further details.

503: Tag followed by weight

Tag may not be followed by weight. Move the location of weight next to a token or a rule.

510: No expression to be repeated

The grammar compiler found repetition brackets following no expression. The repetition brackets must follow tokens or sub-rules to be repeated.

513: Repetition brackets follow alternative operator

The grammar compiler found an alternative operator followed by repetition brackets. The repetition brackets must follow tokens or sub-rules to be repeated.

515: Adjacent repetition brackets

The grammar compiler found two or more subsequent repetition brackets. The repetition brackets must follow tokens or sub-rules to be repeated.

516: Repetition brackets follow option brackets

The grammar compiler found option brackets followed by repetition brackets. The repetition brackets must follow tokens or sub-rules to be repeated.

520: Invalid string within repetition brackets

Repetition number must be specified within repetition brackets, but an invalid string is given.

521: No minimum repetition number

Minimum repetition number is missing within repetition brackets. Maximum repetition number is optional, but minimum repetition number must be specified.

523: Invalid minimum repetition number

Invalid minimum repetition number is found within repetition brackets. The minimum repetition number must be no less than zero.

524: Invalid maximum repetition number

Invalid maximum repetition number is found within repetition brackets.

525: Invalid maximum repetition number

Invalid maximum repetition number is found within repetition brackets. Maximum repetition number must be no greater than 16.

526: Invalid repetition numbers

The minimum repetition number within repetition brackets is greater than the maximum repetition number.

527: No number within repetition brackets

No repetition number is given within repetition brackets.

528: Invalid minimum repetition number

Invalid minimum repetition number is found within repetition brackets. The minimum repetition number is too big.

540: Multiple assignment operators in rule definition

The grammar compiler found multiple assignment operators in a single rule definition. This generally happens when a semicolon of a rule definition is missing.

555: Root rule represents *$VOID*

The root rule of your grammar represents *$VOID*, which cannot be used as a speech recognition grammar. See Section 12 for further details.

560: Same token with different tags

A rule defined in your grammar represents the alternative of the same token or sub-rule, but they have different tags. Remove one of them to resolve this error.

601: Failed to open output file

Sentence generator failed to open output text file.