JSpeech Grammar Format

W3C Note 05 June 2000

This Version:
http://www.w3.org/TR/2000/NOTE-jsgf-20000605
Latest version:
http://www.w3.org/TR/jsgf
Editors:
Andrew Hunt, Speech Works International
<andrew.hunt@speechworks.com>

Abstract

The JSpeech Grammar Format (JSGF) is a platform-independent, vendor-independent textual representation of grammars for use in speech recognition. Grammars are used by speech recognizers to determine what the recognizer should listen for, and so describe the utterances a user may say. JSGF adopts the style and conventions of the Java™ Programming Language in addition to use of traditional grammar notations.

This document is derived from the Java™ Speech API Grammar Format (Version 1.0, October, 1998) which is available from Sun Microsystems's web site: http://java.sun.com/products/java-media/speech/.

Sun Microsystems wishes to submit this document for consideration by the W3C Voice Browser Working Group towards the development of internet standards for speech technology. We expect the resulting W3C recommendations to be of great importance to the developer community.

Please refer to Sun's submission for statements on IP rights.

Status of This Document

This document is a submission to the World Wide Web Consortium from Sun Microsystems, Inc. (see Submission Request, W3C Staff Comment). For a full list of all acknowledged Submissions, please see Acknowledged Submissions to W3C.

This document is a Note made available by W3C for discussion only. This work does not imply endorsement by, or the consensus of the W3C membership, nor that W3C has, is, or will be allocating any resources to the issues addressed by the Note. This document is a work in progress and may be updated, replaced, or rendered obsolete by other documents at any time.

This document is derived from an existing specification published by Sun and developed together with companies that collectively wrote the Java™ Speech API. That specification is known as the Java Speech API Grammar Format, and is available for reference from http://java.sun.com/products/java-media/speech/. Except for the change in the specification's name and the corresponding references in the specification, this document is technically identical to the previously published document. We have changed the name simply to protect Sun trademarks. In any case, we expect that any derived specification produced by the Consortium will have a different name.

Should any changes be required to the document, we would expect future versions to be produced by the W3C process. Sun maintains ownership of the JSGF specification and reserves the right to maintain and evolve the JSGF specification independently and such independent maintenance and evolution shall be owned by Sun.

A list of current W3C technical documents can be found at the Technical Reports page.

Contents

Preface

Scope

The following are treated as programmatic issues that should be handled through the API of the speech recognizer:

- Mechanisms for loading, creating and deleting of grammars in a speech recognizer, activation of grammars for recognition, and other grammar management functionality.
- Conventions for loading grammars: for example, via URLs on a web site.
- Mechanisms for receiving results of recognition for a grammar and processing of those results.
- Error handling for syntactically incorrect grammar definitions.

- Vocabulary management including handling of token pronunciations.

Contributions

Sun Microsystems, Inc. received contributions to this specification from Apple Computer, Inc., AT&T, Dragon Systems, Inc., IBM Corporation, Novell, Inc., Philips Speech Processing and Texas Instruments Incorporated as well as from many internet reviewers.

JSpeech Grammar Format Specification

1. Introduction

Speech recognition systems provide computers with the ability to listen to user speech and determine what is said. Current technology does not yet support *unconstrained* speech recognition: the ability to listen to any speech in any context and transcribe it accurately. To achieve reasonable recognition accuracy and response time, current speech recognizers constrain what they listen for by using *grammars*.

The *JSpeech Grammar Format* (JSGF) defines a platform-independent, vendor-independent way of describing one type of grammar, a *rule grammar* (also known as a *command and control* grammar or *regular grammar*). It uses a textual representation that is readable and editable by both developers and computers, and can be included in source code. The other major grammar type, the *dictation grammar*, is not discussed in this document.

A rule grammar specifies the types of *utterances* a user might say (a spoken utterance is similar to a written sentence). For example, a simple window control grammar might listen for "open a file", "close the window", and similar commands.

What the user can say depends upon the context: is the user controlling an email application, reading a credit card number, or selecting a font? Applications know the context, so applications are responsible for providing a speech recognizer with appropriate grammars.

This document is the specification for the JSpeech Grammar Format. First, the basic naming and structural mechanisms are described. Following that, the basic components of the grammar, the *grammar header* and the *grammar body*, are described. The grammar header declares the *grammar name* and lists the *imported* rules and grammars. The grammar body defines the *rules* of this grammar as combinations of speakable text and references to other rules. Finally, some simple examples of grammar declarations are provided.

1.1 Related Documentation

The following is a list of related documentation that may be helpful in understanding and using the JSpeech Grammar Format.

The JSpeech Grammar Format adopts some of the style and conventions of the Java™ Programming Language. Readers interested in a comprehensive specification are referred to *The Java™ Language Specification*, Gosling, Joy and Steele, Addison Wesley, 1996 (GJS96).

Grammars in the JSpeech Grammar Format may be written with the Unicode character set, as defined in *The Unicode Standard, Version 2.0*, The Unicode Consortium, Addison-Wesley Developers Press, 1996 (Uni96).

---

2. Definitions

2.1 Grammar Names and Package Names

Each grammar defined by JSpeech Grammar Format has a unique name that is declared in the grammar header. Legal structures for grammar names are:

```
packageName.simpleGrammarName
grammarName
```

The first form (package name + simple grammar name) is a *full grammar name*. The second form is a *simple grammar name* (grammar name only). Examples of full grammar names and simple grammar names include:

```
com.sun.speech.apps.numbers
edu.unsw.med.people
examples
```

The package name and grammar name have the same format as packages and classes in the Java™ Programming Language. A full grammar name is a dot-

separated list of *identifiers*[1] (see "The Java Language Specification", Gosling, Joy and Steele, Addison Wesley, 1996, (GJS96) §3.8 and §6.5).

The grammar naming convention also follows the naming convention for classes in the Java Programming Language (see GJS96). The convention minimizes the chance of naming conflicts. The package name should be:

```
reversedDomainName.localPackaging
```

For example, for `com.sun.speech.apps.numbers`, the `com.sun` part is Sun's reversed Internet domain name, `speech.apps` is the local package name for Sun-wide division of the name space, and `numbers` is the simple grammar name.

2.2 Rulenames

A grammar is composed of a set of rules that together define what may be spoken. Rules are combinations of speakable text and references to other rules. Each rule has a unique *rulename*. A reference to a rule is represented by the rule's name in surrounding <> characters (less-than and greater-than).

A legal rulename is similar to an identifier in the Java Programming Language but allows additional extra symbols. A legal rulename is an unlimited-length sequence of Unicode characters matching the following[2]:

- Any character that is legal in an identifier of the Java Programming Language.
- The following additional punctuation symbols:
  + - : ; , = | / \ ( ) [ ] @ # % ! ^ & ~

Grammar developers should be aware of two specific constraints. First, rulenames are compared with exact Unicode string matches, so case is significant. For example, `<Name>`, `<NAME>` and `<name>` are different. Second, whitespace[3] is not permitted in rulenames.

The rulenames `<NULL>` and `<VOID>` are reserved. These special rules are discussed later in this section.

The Unicode character set includes most writing scripts from the world's living languages, so rulenames can be written in Chinese, Japanese, Korean, Thai, Arabic, European languages, and many more. The following are examples of rulenames.

- <hello>
- <Zürich>
- <user_test>
- <$100>

- <1+2=3>
- <π α β>

2.2.1 Qualified and Fully-Qualified Names

Although rulenames are unique within a grammar, separate grammars may reuse the same simple rulename. A later section introduces the `import` statement, which allows one grammar to reference rules from another grammar. When two grammars use the same rulename, a reference to that rulename may be ambiguous. *Qualified names* and *fully-qualified names* are used to reference between grammars without ambiguity.

A fully-qualified rulename includes the *full grammar name* and the simple *rulename*. For example:

```
<com.sun.greetings.english.hello>
<com.sun.greetings.deutsch.gutenTag>
```

A qualified rulename includes only the *simple grammar name* and the *rulename* and is a useful shorthand representation. For example:

```
<english.hello>
<deutsch.gutenTag>
```

The following conditions apply to the use of rulenames:

- Qualified and fully-qualified rulenames may not be used on the left side of the definition of a rule.
- Import statements must use fully-qualified rulenames.
- Local rules can be referenced by qualified and fully-qualified names using the form `<localGrammarName.ruleName>`.

2.2.2 Resolving Rulenames

It is an error to use an ambiguous reference to a rulename. The following defines behavior for resolving references:

- Local rules have precedence. If a local rule and one or more imported rules have the same name, `<ruleName>`, then a simple rule reference to `<ruleName >` is a reference to the local rule.
- If two or more imported rules have the same name, `<ruleName>`, but there is no local rule of the same name, then a simple rule reference to `<ruleName >` is ambiguous and is an error. To resolve this ambiguity these imported rules must be referred to by their qualified or fully-qualified names.
- If two or more imported rules have the same name and come from grammars

with the same simple grammar name (but necessarily different package names), then a simple rule reference or qualified rule reference is ambiguous and is an error. These imported rules must be referred to by their fully-qualified names.

- A reference by a fully-qualified rulename is never ambiguous.

When a rulename reference cannot be resolved (not defined locally and not a public rule of an imported grammar), the handling of the reference is defined by the recognizer's software interface[4].

2.2.3 Special Rulenames

The JSpeech Grammar Format defines two special rules, <NULL> and <VOID>. These rules are universally defined - they are available in any grammar without an explicit import statement - and they cannot be redefined. Both names are fully-qualified so no qualifying grammar name is required.

<NULL> defines a rule that is automatically matched: that is, matched without the user speaking any word.

<VOID> defines a rule that can never be spoken. Inserting <VOID> into a sequence automatically makes that sequence unspeakable[5].

The <NULL> and <VOID> rules are typically used in specialized circumstances. They can be used to block and enable parts of grammars, to control recursion, and to perform other advanced tasks. The *Uses of <NULL> and <VOID>* are described later in this document.

2.3 Tokens

A *token*, sometimes called a *terminal symbol*, is the part of a grammar that defines what may be spoken by a user. Most often, a token is equivalent to a *word*. Tokens may appear in isolation, for example,

```
hello
konnichiwa
```

or as sequences of tokens separated by whitespace characters, for example,

```
this is a test
open the directory
```

In the JSpeech Grammar Format, a token is a character sequence bounded by whitespace, by quotes or delimited by the other symbols that are significant in the grammar:

```
; = | * + <> () [] {} /* */ //
```

A token is a reference to an entry in a *recognizer's vocabulary*, often referred to as the *lexicon*. The recognizer's vocabulary defines the *pronunciation* of the token. With the pronunciation, the recognizer is able to listen for that token.

The JSpeech Grammar Format allows *multi-lingual* grammars, that is, grammars that include tokens from more than one language. However, most recognizers operate *mono-lingually* so a typical grammar will contain only one language. It is the responsibility of the application that loads a grammar into a recognizer to ensure that it has appropriate language support. As an example, the following is a simple multi-lingual rule.

```
<no> = no | nein | nao | non | nem;
```

Most recognizers have a comprehensive vocabulary for each language they support. However, it is never possible to include 100% of a language. For example, names, technical terms and foreign words are often missing from the vocabulary. For tokens missing from the vocabulary, there are three possibilities:

- An application or user can add the token and pronunciation to the recognizer's vocabulary to ensure consistent recognition.
- Good recognizers are able to guess the pronunciation of many words not in the vocabulary.
- If neither of the previous points apply, the behavior is determined by the software interface of the recognizer. In most cases, an undefined token will be unspeakable (equivalent to `<VOID>`), or it will cause an error or exception.

Tokens do not need to be normal written words of a language, assuming that the token is properly defined in the recognizers vocabulary. For example, to handle the two pronunciations of "read" (past tense sounds like "red", present tense sounds like "reed") an application could define two separate tokens "read_past" and "read_present" with appropriate pronunciations.

2.3.1 Quoted Tokens

A token does not need to be a word. A token may be a sequence of words or a symbol. Quotes can be used to surround multi-word tokens and special symbols. For example:

```
the "New York" subway
"+"
```

A multi-word token is useful when the pronunciation of words varies because of the context. Multi-word tokens can also be used to simplify the processing of results, for example, getting single-token results for "New York", "Sydney" and "Rio de Janeiro."

Quoted tokens can be included in the recognizer's vocabulary like any other token.

If a multi-word quoted token is not found in the vocabulary, then the default behavior is to determine the pronunciation of each space-separated token within the quotes, but otherwise treat the text within quotes as a single token.

To include a quote symbol in a token, surrounding quotes must be used and the quote within the token must be preceded by a backslash `\'. Similarly, to include a backslash in a quoted token, it should be preceded by another backslash. For example, the following are two tokens representing a single backslash and a single quote character.

```
"\\" "\""
```

White space is significant in quoted tokens.

2.3.2 Symbols and Punctuation

Most speech recognizers provide the ability to handle common symbols and punctuation forms. For example, recognizers for English usually handle apostrophes ("Mary's", "it's") and hyphens ("new-age").

There are, however, many textual forms that are difficult for a recognizer to handle unambiguously. In these instances, a grammar developer should use tokens that are as close as possible to the way people will speak and that are likely to be built into the vocabulary. The following are common examples.

- Numbers: "0 1 2" should be expanded to "zero one two" or "oh one two". Similarly, "call 555 1234" should be expanded to "call five five five one two three four."
- Dates: "Dec 25, 1997" should be written as "December twenty fifth nineteen ninety seven."
- Abbreviations and acronyms: "Mr." should be written as "mister", and "St" should be written as "street".
- Special symbols: `&' as "ampersand" or "and", `+' as "plus", and so on.

2.4 Comments

Comments may appear in both the header and body. The comment style of the Java Programming Language is adopted (see GJS96). There are two forms of comment.

/* text */	A traditional comment. All text between /* and */ is ignored.

// text	A single-line comment. All the text from // to the end of a line is ignored.

Comments do not nest. Furthermore, `//` has no special meaning within comments beginning with `/*`. Similarly, `/*` has no special meaning within comments beginning with `//`.

Comments may appear anywhere in a grammar definition except within tokens, quoted tokens, rulenames, tags and weights.

The JSpeech Grammar Format supports *documentation comments* with a similar style to the documentation comments of the Java Programming Language (GJS96, §18). These special comments are defined in the section on *Documentation Comments*.

---

3. Grammar Header

A single file defines a single grammar. The definition grammar contains two parts: the *grammar header* and the *grammar body*. The grammar header includes a self-identifying header, declares the name of the grammar and declares imports of rules from other grammars. The body defines the rules of the grammar, some of which may be public.

3.1 Self-Identifying Header

A JSpeech Grammar Format document starts with a self-identifying header. This header identifies that the document contains JSGF and indicates the version of JSGF being used (currently "V1.0"). Next, the header optionally specifies the character encoding used in the document. The header may also optionally specify the locale of the grammar specified in the document. The locale[6] specifies the language and optionally the country or regional variant that the grammar supports. The header is terminated by a semi-colon character and a newline character.

The header format is:

```
#JSGF version char-encoding locale;
```

The following are examples of self-identifying headers.

```
#JSGF V1.0;
#JSGF V1.0 ISO8859-5;
```

```
#JSGF V1.0 JIS ja;
```

The first example does not provide a character encoding or locale, so a reasonable default would be assumed. In the US, the default might be ISO8859-1 (a standard character set) and "en" (the symbol for English). The second example defines the ISO8859-5 set (Cyrillic) but the default locale is assumed. The final example defines the "JIS" character set (one of the Japanese character sets) and defines the language as "ja" (Japanese).

The hash character (`#') must be the first character in the document and all characters in the self-identifying header must be in the ASCII subset of the encoding being used.

3.2 Grammar Name Declaration

The grammar's name must be declared as the first statement of that grammar. The declaration must use the full grammar name (package name + simple grammar name). Thus, the declaration format is either of the following:

```
grammar packageName.simpleGrammarName;
grammar grammarName;
```

The naming of packages and grammars is described in the section on *Grammar Names and Package Names*.

For example:

```
grammar com.sun.speech.apps.numbers;
grammar edu.unsw.med.people;
grammar examples;
```

3.3 Import

The grammar header can optionally include *import* declarations. The import declarations follow the grammar declaration and must come before the grammar body (the rule definitions). An import declaration allows one or all of the public rules of another grammar to be referenced locally. The format of the import statement is one of the following

```
import <fullyQualifiedRuleName>;
import <fullGrammarName.*>;
```

For example,

```
import <com.sun.speech.app.index.1stTo31st>;
import <com.sun.speech.app.numbers.*>;
```

The first example is an import of a single rule by its fully-qualified rulename: the rule `<1stTo31st>` from the grammar `com.sun.speech.app.index`. The imported rule, `<1stTo31st>`, must be a public rule of the imported grammar.

The use of the asterisk in the second import statement requests the import of all public rules of the `numbers` grammar. For example, if that grammar defines three public rules, `<digits>`, `<teens>`, `<zeroToMillion>`, then all three may be referenced locally.

Note that because both a grammar name and a rulename or asterisk are required, the import statement is never of the form:

```
import <ruleName>; // not legal
```

An imported rule can be referenced in three ways: by its simple rulename (e.g. `<digits>`), by its qualified rulename (e.g. `<numbers.digits>`), or by its fully- qualified rulename (`<com.sun.speech.apps.numbers.digits>`).

The name resolving behavior is defined earlier in this document in *Resolving Rulenames*. Note that an import statement is optional when an imported rule is always referenced by its fully-qualified rulename.

```
// Importing com.sun.speech.app.numbers is optional
<rule> = <com.sun.speech.app.numbers.digits>;
```

---

4. Grammar Body

4.1 Rule Definitions

The grammar body defines *rules*. Each rule is defined in a *rule definition*. A rule is defined once in a grammar. The order of definition of rules is not significant[7].

The two patterns for rule definitions are:

```
<ruleName> = ruleExpansion ;
public <ruleName> = ruleExpansion ;
```

The components of the rule definition are (1) an optional *public* declaration, (2) the name of the rule being defined, (3) an equals sign `=', (4) the *expansion* of the rule, and (5) a closing semi-colon `;'.

White space is ignored before the definition, between the public keyword and the rulename, around the equal-sign character, and around the semi-colon. White space is significant within the rule expansion.

The *rule expansion* defines how the rule may be spoken. It is a logical combination of *tokens* (text that may be spoken) and *references* to other rules. The term "expansion" is used because an expansion defines how a rule is expanded when it

is spoken - a single rule may expand into many spoken words plus other rules which are themselves expanded. Later sections define the legal expansions.

4.1.1 Public Rules

Any rule in a grammar may be declared as public by the use of the `public` keyword. A public rule has three possible uses:

- It can be referenced within the rule definitions of another grammar by its fully-qualified name or with an import declaration and an unambiguous reference form.
- It can be used as an *active* rule for recognition. That is, the rule can be used by a recognizer to determine what may be spoken.
- It can be referenced locally: that is, by any public or non-public rule defined in the same grammar.

Without the public declaration, a rule is implicitly *private*[8] and can only be referenced within rule definitions in the local grammar.

4.2 Rule Expansions

The simplest rule expansions are a reference to a token and a reference to a rule. For example,

```
<a> = elephant;
<b> = <x>;
<c> = <com.acme.grammar.y>;
```

The rule <a> expands to a single token "elephant". Thus, to speak <a> the user must say the word "elephant".

The rule <b> expands to <x>. This means that to speak <b>, the user must say something that matches the rule <x>. Similarly, to speak rule <c> the user must speak something that matches the rule <com.acme.grammar.y>[9].

In more formal terms, the following are legal expansions.

- Any token.
- A reference to any public or non-public rule defined in the same grammar.
- A reference to any public rule of another grammar which has been imported into the current grammar.
- A reference to a public rule of another grammar when referenced with its fully-qualified rulename (with or without an import).

These reference policies are defined locally, that is, by the scope of the current rule. For example, a rule in `grammar1` can legally reference a public rule of `grammar2`, which in turn references a non-public rule of `grammar2`. In other words, a rule definition can indirectly reference a private rule of another grammar through a public rule of the other grammar.

An empty definition is not legal.

```
<d> = ; // not legal
```

Definition of a rule as either `<NULL>` or `<VOID>` is legal.

```
<e> = <NULL>; // legal
<f> = <VOID>; // legal
```

The following sections explain ways in which more complex rules can be defined by logical combinations of legal expansions using:

- *Composition*: sequences of expansions and sets of alternative expansions.
- *Grouping* using parentheses and brackets.
- *Unary operators* for repetition of expansions.
- Attachment of application-specific *tags* to expansions.

4.3 Composition

4.3.1 Sequences

A rule may be defined by a *sequence* of expansions. A sequence of legal expansions, each separated by white space, is itself a legal expansion. For example, because both tokens and rule references are legal expansions, the following are legal rule definitions.

```
<where> = I live in Boston;
<statement> = this <object> is <OK>;
```

To speak a sequence, each item in the sequence must be spoken in the defined order. In the first example, to say the rule `<where>`, the speaker must say the words "I live in Boston" in that exact order. The second example mixes tokens with references to the rules `<object>` and `<OK>`. To say the rule `<statement>`, the user must say "this" followed by something which matches `<object>`, then "is", and finally something matching `<OK>`.

The items in a sequence may be any legal expansion. This includes the structures described below for alternatives, groups and so on.

4.3.2 Alternatives

A rule may be defined as a *set of alternative* expansions separated by vertical bar characters `|' and optionally by whitespace. For example:

```
<name> = Michael | Yuriko | Mary | Duke | <otherNames>;
```

To say the rule `<name>`, the speaker must say one, and only one, of the items in the set of alternatives. For example, a speaker could say "Michael", "Yuriko", "Mary", "Duke" or anything that matches the rule `<otherNames>`. However, the speaker could not say "Mary Duke".

Sequences have higher precedence than alternatives. For example,

```
<country>= South Africa | New Zealand | Papua New Guinea;
```

is a set of three alternatives, each naming a country.

An empty alternative is not legal.

```
<name> = Michael | | Mary; // not legal
<name> = Michael | Mary | ; // not legal
```

4.3.3 Weights

Not all ways of speaking a grammar are equally likely. Weights may be attached to the elements of a set of alternatives to indicate the likelihood of each alternative being spoken. A weight is a floating point number[10] surrounded by forward slashes, e.g. /3.14/. The higher the weight, the more likely it is that an entry will be spoken. The weight is placed before each item in a set of alternatives. For example:

```
<size> = /10/ small | /2/ medium | /1/ large;
<color> = /0.5/ red | /0.1/ navy blue | /0.2/ sea green;
<action> = please (/20/save files |/1/delete all files);
<place> = /20/ <city> | /5/ <country>;
```

The weights should reflect the occurrence patterns of the elements of a set of alternatives. In the first example, the grammar writer is indicating that "small" is 10 times more likely to be spoken than "large" and 5 times more likely than "medium."

The following conditions must be met when specifying weights:

- If a weight is specified for one item in a set of alternatives, then a weight must be specified for every item (the "all or nothing rule").
- Weights are floating point numbers. For example, 56, 0.056, 3.14e3, 8f.
- Only a floating point number and whitespace is allowed within the slashes.
- Weights must be zero or greater. A zero weight indicates that the item can never be spoken: equivalent to replacing the item by `<VOID>`. (Zero weights

are useful in developing grammars.)

- At least one non-zero positive weight is required.

Appropriate weights are difficult to determine and guessing weights does not always improve recognition performance. Effective weights are usually obtained by study of real speech and textual data.

Not all recognizers utilize weights in the recognition process. However, as a minimum, a recognizer is required to ensure that any alternative with a weight of zero cannot be spoken.

4.4 Grouping

4.4.1 (Parentheses)

Any legal expansion may be explicitly grouped using matching parentheses `()'. Grouping has high precedence and so can be used to ensure the correct interpretation of rules. It is also useful for improving clarity. For example, because sequences have higher precedence than alternatives, parentheses are required in the following rule definition so that "please close" and "please delete" are legal.

```
<action> = please (open | close | delete);
```

The following example shows a sequence of three items, with each item being a set of alternatives surrounded by parentheses to ensure correct grouping.

```
<command> = (open | close) (windows | doors)
(immediately  |  later);
```

To say something matching `<command>`, the speaker must say one word from each of the three sets of alternatives: for example, "open windows immediately" or "close doors later".

If a grouping surrounds a single expansion, then the entity is defined to be a sequence of one item[11]. For example:

```
( start )
( <end> )
```

Empty parentheses are not legal.

```
( ) // not legal
```

4.4.2 [Optional Grouping]

Square brackets may be placed around any rule definition to indicate that the contents are optional. In other respects, it is equivalent to parentheses for grouping and has the same precedence.

For example,

```
<polite> = please | kindly | oh mighty computer;
public <command> = [ <polite> ] don't crash;
```

allows a user to say "don't crash" and to optionally add one form of politeness such as "oh mighty computer don't crash" or "kindly don't crash".

Empty brackets are not legal.

```
[ ] // not legal
```

4.5 Unary Operators

There are three *unary operators* in the JSpeech Grammar Format: the Kleene star, the plus operator and tags. The unary operators share the following features:

- A unary operator may be attached to any legal rule expansion.
- They have high precedence: they attach to the immediate preceding rule expansion.
- Only one unary operator can be attached to any rule expansion (there is an exception to this rule for tags).
- White space between the rule expansion and attached operator is ignored.

Because the precedence of unary operators is higher than that of sequences and alternatives, parentheses must be used to surround a sequence or set of alternatives to attach an operator to the entire entity.

4.5.1 * Kleene Star

A rule expansion followed by the asterisk symbol indicates that the expansion may be spoken *zero or more times*. The asterisk symbol is known as the Kleene star (after Stephen Cole Kleene, who originated the use of the symbol). For example,

```
<command> = <polite>* don't crash;
```

allows a user to say things like "please don't crash", "oh mighty computer please please don't crash", or to ignore politeness with "don't crash".

As a unary operator, this symbol has high precedence. For example, in

```
<song> = sing New York *;
```

the operator applies to the most immediate preceding legal expansion: the token "York". Thus to speak `<song>` the user may say "sing New York", "sing New" and "sing New York York York", but not "sing New York New York". Quotes or

parentheses can be used to modify the scope of the * operator. For example,

```
<song> = sing (New York) *;
```

does match "sing New York New York".

4.5.2 + Plus Operator

A rule expansion followed by the plus symbol indicates the expansion may be spoken *one or more times*. For example,

```
<command> = <polite>+ don't crash;
```

requires at least one form of politeness. So, it allows a user to say "please please don't crash". However, "don't crash" is not legal.

4.6 Tags

Tags provide a mechanism for grammar writers to attach application-specific information to parts of rule definitions. Applications typically use tags to simplify or enhance the processing of recognition results.

Tag attachments do not affect the recognition of a grammar. Instead, the tags are attached to the result object returned by the recognizer to an application. The software interface of the recognizer defines the mechanism for providing tags[12].

A tag is a unary operator. As such it may be attached to any legal rule expansion. The tag is a string delimited by curly braces `{}'. All characters within the braces are considered a part of the tag, including white-space. Empty braces "{}" are a legal tag. In this instance the tag should be interpreted as a zero-length string ("" in the Java Programming Language).

To include a closing brace character in a quote, it must be *escaped* with a backslash `\' character. To include a backslash character, it must also be escaped by a backslash. For example,

```
{ {nasty \\looking\\ tag\} }
```

is processed as the following string:

```
" {nasty \looking\ tag} "
```

The tag attaches to the immediate preceding rule expansion (intervening whitespace is ignored). For example:

```
<rule> = <action> {tag in here};
<command>= please (open {OPEN} | close {CLOSE}) the file;
<country> = Australia {Oz} | (United States) {USA} |
America  {USA}  | (U S of A) {USA};
```

As a unary operator, tag attachment has higher precedence than sequences and alternatives. For example, in

```
<action> = book | magazine | newspaper {thing};
```

the "thing" tag is attached only to the "newspaper" token. Parentheses may be used to modify tag attachment:

```
<action> = (book | magazine | newspaper) {thing};
```

Unlike the other unary operators, more than one tag may follow a rule expansion. For example,

```
<OKRule> = <action> {tag1} {tag2} {tag3}; // legal
```

This is interpreted as if parentheses were used:

```
<OKRule> = ((<action> {tag1}) {tag2}) {tag3}; // legal
```

This syntactic convenience is only permitted for tags. The following are not permitted:

```
<badRule> = <action> * {tag1}; // not legal
<badRule> = <action> {tag1} +; // not legal
```

4.6.1 Using Tags

Tags can simplify the writing of applications by simplifying the processing of recognition results. The content of tags, and the use of tags are entirely up to the discretion of the developer.

One important use of tags is in internationalizing applications. The following are examples of rule definitions for four grammars, each for a separate language. The application loads the grammar for the language spoken by the user into an appropriate recognizer. The tags remain the same across all languages and thus simplify the application software that processes the results. Typically, the grammar name will include the language identifier so that it can be programmatically located and loaded.

In the English grammar:

```
<greeting>= (howdy | good morning) {hi};
```

In the Japanese grammar:

```
<greeting>= (ohayo | ohayogozaimasu) {hi};
```

In the German grammar:

```
<greeting>= (guten tag) {hi};
```

In the French grammar:

```
<greeting>= (bon jour) {hi};
```

4.7 Precedence

The following defines the relative precedence of the entities of a rule expansion in the JSpeech Grammar Format. The list proceeds from highest to lowest precedence.

1. Rulename contained within angle brackets, and a quoted or unquoted token.
2. `()' parentheses for grouping and `[]' for optional grouping.
3. The unary operators (`+', `*', and tag attachment) apply to the tightest immediate preceding rule expansion. (To apply them to a sequence or to alternatives, use `()' or `[]' grouping.)
4. Sequence of rule expansions.
5. `|' separated set of alternative rule expansions.

4.8 Recursion

Recursion is the definition of a rule in terms of itself. Recursion is a powerful tool that enables representation of many complex grammatical forms that occur in spoken languages. Recognizers supporting the JSpeech Grammar Format allow *right recursion*. In right recursion, the rule refers to itself as the last part of its definition. For example:

```
<command> = <action> | (<action> and <command>);
<action> = stop | start | pause | resume | finish;
```

allows the following commands: "stop", "stop and finish", "start and resume and finish".

Nested right recursion is also permitted. Nested right recursion is a definition of a rule that references another rule that refers back to the first rule with each recursive reference being the last part of the definition. For example,

```
<X> = something | <Y>;
<Y> = another thing <X>;
```

Nested right recursion may occur across grammars. However, this is strongly discouraged, as it introduces unnecessary complexity and potential maintenance problems.

Any right recursive rule can be re-written using the Kleene star `*' and/or the plus operator `+'. For example, the following rule definitions are equivalent:

```
<command> = <action> | (<action> and <command>);
<command> = <action> (and <action>) *;
```

Although it is possible to re-write right recursive grammars using the `+' and `*' operators, the recursive form is permitted because it allows simpler and more elegant representations of some grammars. Other forms of recursion (left recursion, embedded recursion) are not supported because the re-write condition cannot be guaranteed[13].

4.9 Uses of <NULL> and <VOID>

The two *Special Rulenames* defined earlier in this document - `<NULL>` and `<VOID>` - have a number of advanced uses.

A reference to `<NULL>` as an alternative to any rule expansion is the same as using the optional grouping. So, the following are equivalent definitions:

```
<x> = a | <NULL>;
<x> = [ a ];
```

The Kleene star and right recursion can be mapped as follows:

```
<x> = <NULL> | a <x>;
<x> = a*;
```

For the two previous cases, the grammars are identical in the sense that the user may speak exactly the same utterances. However, there may be programmatic differences in the representation of results produced by a recognizer when the user says something matching the grammar.

A final use of `<NULL>` and `<VOID>` is for gating. To turn on and off particular parts of a grammar an application could define the rule `<gate>`, place it before the parts of the grammar that will be turned on and off, and alternate the definition of `<gate>` between:

```
<gate> = <NULL>;
<gate> = <VOID>;
```

When `<gate>` is `<VOID>`, it is unspeakable and the grammar region is turned off.

As a caution, to work effectively, this technique requires a recognizer that can efficiently re-compile the grammar after changing the definition of `<gate>`. Not all recognizers perform this re-compilation efficiently.

4.10 Documentation Comments

The JSpeech Grammar Format supports *documentation comments* with a similar style to the documentation comments of the Java Programming Language (GJS96, §18). Such comments can appear before the grammar declaration, before import statements and before the definition of any rule. Hypertext web pages and

other forms of documentation can be automatically generated from these comments.

A documentation comment commences with the characters `/**` and is terminated by `*/`. On each of the comment lines leading `*` characters and any preceding white space is ignored.

The first sentence of the comment (terminated by the first period character) should be a *summary sentence* with a concise description of the entity being commented.

Unlike documentation comments in the Java Programming Language, documentation comments in the JSpeech Grammar Format do not allow HTML tags. This is because HTML tags and JSGF rulename references use the same format. For example, is `<font>` a rulename or an HTML code?

A *tagged paragraph* is marked by a line of a documentation comment that begins with the @ character followed by one of the keywords defined below. A tagged paragraph includes the following lines of the comment up to the start of the next tagged paragraph or to the end of the documentation comment. (Documentation tags are not related to tags in rule definitions.)

The following example shows the basic structure of a documentation comment. *Example 3: Documentation Comments* shows an example of comments for a simple grammar.

```
/**
    * Initial sentence provides a summary.
    * Additional text and paragraphs can be inserted.
    *
    * @tag several types of tag are defined below
    */
```

4.10.1 The @author Tag

The `@author` tag may be used in the documentation comment for the grammar declaration. Although there is no defined structure for the `@author` tag, it is recommended that each author be listed in a separate tag. Multiple `@author` tags may be included in a single comment.

For example,

```
@author James Bond
@author Jose Alvarez
```

4.10.2 The @version Tag

A single `@version` tag may be included in the documentation comment of a grammar declaration. There is no defined structure or format. For example,

```
@version versionInformation
```

4.10.3 The @see Tag

Any number of `@see` tags may be used in any documentation comment. The tag indicates a cross-reference to another rule (local or imported) or to another grammar. The `.*` suffix indicates the reference is to a grammar.

```
@see <com.acme.number.zeroToTen>
@see <com.acme.number.*>
@see <zeroToTen>
```

4.10.4 The @example Tag

The `@example` tag can be provided in documentation comments for rule declarations to provide an example of how the rule may be spoken. Appropriate examples make grammars and rules easier to understand. The example text may also be used by grammar tools to verify the correctness of the grammar.

```
@example please say your name and age
@example I love "south america"
```

Developers are encouraged to use the same tokenization in examples as used in the rule definition. For example, the second sample above should be interpreted as two tokens because of the quotes around "south america". However, tool developers should consider the possibility that the example text includes human-readable formatting for clarity. For English this might include punctuation (period, comma, question mark, exclamation point etc.), capitalization of some tokens, modified tokenization (e.g. missing quotes).

The `@example` tag may include rulename references. For example,

```
@example I want a pizza with <topping>
```

would expand out with each example tag defined for the `<topping>` rule ("I want a pizza with pepperoni", "I want a pizza with mushrooms" and so on).

---

5. Examples

By combining simple rules together, it is possible to build complex grammars that capture what users can say. The following are examples of grammars with

complete headers and bodies.

5.1 Example 1: Simple Command and Control

This example shows two basic grammars that define spoken commands that control a window. Optional politeness is included to show how speech interaction can be made a little more natural and conversational.

```
#JSGF V1.0;

grammar com.acme.politeness;

// Body
public <startPolite> = (please | kindly | could you |
oh  mighty  computer) *;
public <endPolite> = [ please | thanks | thank you ];
```

The `politeness` grammar is not useful on its own but is imported into the `commands` grammar to add the conversational style.

```
#JSGF V1.0 ISO8859-1 en;

grammar com.acme.commands;
import <com.acme.politeness.startPolite>;
import <com.acme.politeness.endPolite>;

/**
  * Basic command.
  * @example please move the window
  * @example open a file
  */

public <basicCmd> = <startPolite> <command> <endPolite>;

<command> = <action> <object>;
<action> = /10/ open |/2/ close |/1/ delete |/1/ move;
<object> = [the | a] (window | file | menu);
```

The `commands` grammar defines a single `public` rule, `<basicCommand>`, which is composed of two imported rules, `<startPolite>` and `<endPolite>`, and three private rules, `<command>`, `<action>` and `<object>`. Both the `<action>` and `<object>` rules are sets of alternative words and the actions list has weights that indicate that "open" is more likely than the others. The `<command>` rule defines a sequence in which `<action>` is followed optionally by "the" or "a" and always by an `<object>`.

Because `<com.acme.commands.basicCommand>` is public, it can be made active for recognition. When it is active, users may say commands such as:

- "open a window"

- "close file please"
- "oh mighty computer please open a menu"

5.2 Example 2: Resolving Names

The following example grammar illustrates the use of fully-qualified names for an application that deals with clothing. The two imported grammars define import rules regarding pants and shirts, including the lists of colors that shirts and pants may have. The local `<color>` rule is a combination of the imported color lists. Because a reference to `<color>` is ambiguous (it could come from either the pants or shirts grammar), qualified or fully-qualified names are required.

```
#JSGF V1.0;
grammar com.acme.selections;
import <com.acme.pants.*>;
import <com.sun.shirts.*>;

<color> = <com.acme.pants.color> |
<com.acme.shirts.color>;
public <statement> = I like <color>;
```

The reference to `<color>` in the last definition is not ambiguous: because local rules have precedence over imported rules, it refers to the locally-defined `<color>` rule. In the definition of the local `<color>` rule, qualified names could have been used as they would be unambiguous references: that is,

```
<color> = <pants.color> | <shirts.color>;
```

5.3 Example 3: Documentation Comments

The following example grammar illustrates the use of documentation comments for a simple grammar. It shows all three types of use of documentation comments, for the grammar declaration, for import statements and for rule definitions.

```
#JSGF 1.0 ISO8559-1;

/**
  * Define simple travel directives.
  *
  * @author Mary Contrary
  * @version 3.141beta
  */

grammar com.acme.travel;

/**
  * Get a list of city names: <city>.
  */
```

```
import <com.acme.cities.*>;

/**
  * A simple travel command
  *
  * @example go from sydney to tokyo to dublin
  * @example go from "san francisco" to bangkok
  */

public <travel> = go from <city> ( to <city> )+;
```

Footnotes

[1] An identifier in the Java™ Programming Language is an unlimited-length sequence of Unicode characters. The first character is a letter or one of a set of special symbols (including `$' and `_'). Following characters include letters, numbers, the special symbols and other characters.

[2] An API method is provided to test the legality of rulenames.

[3] The set of whitespace characters is defined by the Unicode character set.

[4] Default behavior should be to throw an exception or issue an error message.

[5] An API should define static variables for NULL and VOID for efficient testing.

[6] See RFC 1766.

[7] These properties differ from some linguistic and computational grammar formats which permit multiple alternate definition of non-terminals or for which order is significant.

[8] Unlike the Java Programming Language, the JSpeech Grammar Format does not have keywords for `private` or `protected`.

[9] As defined in the section on *Rulenames*, if `<x>` or `<com.acme.grammar.y>` are imported rules, there should be an appropriate import statement and the rule must be declared public in the imported grammar.

[10] The Java Programming Language requires an `f' or `F' suffix for floating point numbers. This is not required in the JSpeech Grammar Format.

[11] This definition provides consistency in parsing and other grammar analysis.

[12] The minimum expected behavior is that a parse tree should include the tags.

[13] Technically speaking, the JSpeech Grammar Format defines what is formally called a *regular grammar*. Some features typically associated with a *context-free grammar* are permitted for clarity or convenience.
