Java Speech Grammar Format Specification

Version 0.5 — August 28, 1997

Grammars are used by speech recognizers to determine what the recognizer should listen for and to describe what utterances a user may say. The Java™ Speech Grammar Format is a platform-independent, vendor-independent textual representation of grammars in the flavor of Java that is readable and editable by both developers and computers.

THE NETWORK IS THE COMPUTER™

Contents

-

Notes to Reviewers

This document describes the Java™ Speech Grammar Format (JSGF) and explains how it can be used to define cross-platform, cross-vendor recognition grammars. This specification is an extract from the Java Speech Application Programming Interface (JSAPI) specification that will be released later in 1997. When the full specification is released, the Java Speech Grammar Format specification will be included as part of the programming guide.

Review Comments

We are very interested in your input concerning the Java Speech Grammar Format specification. Send your comments to:

```
javaspeech-comments@sun.com
```

Please be sure to include the version number and date of the document you are reviewing with your comments. We anticipate releasing a small number of updates to our documentation during the review period. These updates will incorporate responses to comments. The earlier we receive your feedback, the more likely it will be taken into consideration for the next update.

Because of the high level of interest in the Java Speech API, the Java Speech Grammar Format and the Java Synthesis Markup Language, we are unable to respond directly to individual comments or questions, but we will carefully read and evaluate all of the input we receive.

JSGF and JSAPI

This specification for the Java Speech Grammar Format defines a textual representation of the recognition grammars but does not address the issues listed below. These programmatic issues are covered in the documentation for the Java Speech API which is expected to be released later in 1997.

- Mechanisms for loading and deleting of grammars in a speech recognizer, activation of grammars for recognition, and other grammar management functionality.
- Mechanisms for receiving results of recognition for a grammar and processing of those results.
- Specific error handling behavior for undefined and ambiguously defined rule references.
- Vocabulary management such as provision of token pronunciations.

Issues for this Release

Many aspects of the Java Speech Grammar Format are fully specified. However, some areas are still under development. Reviewers are especially encouraged to provide feedback in these areas.

Specification Issues

The following areas in the Java Speech Grammar Format are either not yet defined or are not fully defined in this version of the specification:

- A formal syntactic specification of JSGF.
- *Format header*: an optional header may be introduced to improve support for multi-lingual applications. It could define the JSGF version used in the document, the character encoding and possibly the language. e.g.
  `#jsgf 1.0 ISO-8859-1 en.us;`
- *"Pragma"* facility for specifying grammar-specific parameters.
- *Special symbols*:
  `<NULL>` and empty groups () [] that are skipped.
  `<GARBAGE>` matching any spoken input for word and phrase spotting.
  `<UNSPEAKABLE>` which cannot be matched to any speech.
- *Unicode symbols* with Java-style reference '\u003C'.

Plans for Future Releases

Sun and its partners are considering new capabilities and features that may appear in a future release of the Java Speech Grammar Format specification. Features that we are considering for future releases include:

- *Codebase conventions* for locating grammars as URLs.

Comments regarding the priority of these features, or other new features that you would like to see, are appreciated.

Web Resources

To obtain information about the Java Speech API, see the web site at:

```
http://java.sun.com/products/java-media/speech/
```

To obtain information about other Java Media and Communications APIs, see the web site at:

```
http://java.sun.com/products/java-media/
```

Mailing Lists

Discussion lists have been set up for anyone interested in the Java Speech API, the Java Speech Grammar Format specification, and the Java Synthesis Markup Language. The `javaspeech-announce` mailing list will carry important announcements about releases and updates. The `javaspeech-interest` mailing list is for open discussion of the Java Speech API and the associated specifications.

To subscribe to the `javaspeech-announce` list or the `javaspeech-interest` list, send email with `subscribe javaspeech-announce` or `subscribe javaspeech-interest` or both in the message body to:

```
javamedia-request@sun.com
```

The `javaspeech-announce` mailing list is moderated. It is not possible to send email to that list.

To send messages to the interest list, send email to:

```
javaspeech-interest@sun.com
```

To unsubscribe from the `javaspeech-announce` list or the `javaspeech-interest` list, send email with `unsubscribe javaspeech-announce` or `unsubscribe javaspeech-interest` or both in the message body to:

```
javamedia-request@sun.com
```

Revision History

Version 0.5: First public Beta release.

Contributions

Sun Microsystems, Inc. has worked in partnership with leading speech technology companies to define the specifications for the Java Speech API and the Java Speech Grammar Format (JSGF). These companies bring decades of research on speech technology and experience in the development and use of speech applications. Sun is grateful for the contributions of:

- Apple Computer, Inc.
- AT&T
- Dragon Systems, Inc.
- IBM Corporation
- Novell, Inc.
- Philips Speech Processing
- Texas Instruments Incorporated

Java Speech Grammar Format Specification

1.0 Introduction

Speech recognition systems provide computers with the ability to listen to a user's speech and determine what they say. 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 *Java™ Speech 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). It uses a textual representation that is readable and editable by both developers and computers, and can be included in Java 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 defines the Java Speech 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 Java Speech Grammar Format.

The Java Speech Grammar Format has been developed for use with recognizers that implement the Java Speech API. However, it may also be used by other speecch recognizers and in other types of applications.

Readers interested in the programmatic use of the Java Speech Grammar Format with the Java Speech API are referred to the technical documentation and the Programmers Guide for the API. That guide is scheduled for public release by the end of 1997. Among other issues, those documents define the mechanisms for loading grammars into recognizers, methods for controlling and modifying loaded grammars, error handling and so on.

The Java Speech Grammar Format has adopted some of the style and conventions of the Java Programming Language. There are dozens of books that describe Java programming. Readers interested in a comprehensive specification are referred to *The Java Language Specification*, Gosling, Joy and Steele, Addison Wesley, 1996 (GJS96).

Finally, like the Java Programming Language, the Java Speech Grammar Format is defined with the Unicode character set. The full specification is defined in *The Unicode Standard, Version 2.0*, The Unicode Consortium, Addison-Wesley Developers Press, 1996 (Uni96).

2.0 Definitions

2.1 Grammar Names and Package Names

Each grammar defined by Java Speech Grammar Format has a unique name that is declared in the grammar header. The structure of these names is either of the following:

```
packageName.grammarName
grammarName
```

The first pattern (package name + grammar name) is a *full grammar name* and the second 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 *Java identifiers*[1] (see 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`, `COM.Sun` is the reversed Internet domain name, `speech.apps` is the local package name to divide the name space internally, and finally `numbers` is the name of the grammar.

2.2 Rule Names

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

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

- Characters matching `java.lang.Character.isJavaIdentifierPart` including the Unicode letters and numbers plus other symbols.
- The following additional punctuation symbols:

1. A Java identifier 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. The `java.lang.Character` class defines methods to test identifiers and document the character sets in more detail: `isJavaIdentifierStart` and `isJavaIdentifierPart`.

2. The Java Speech API will define a method to test the set of legal characters in rule names.

+ - : ; , = | / \ ( ) [ ] @ # % ! ^ & ~

Grammar developers should be aware of two constraints. First, rule names 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 rule names.

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

```
<hello>

<Zürich>

<user_test>

<$100>

<1+2=3>

<παβ>
```

2.2.1 Qualified and Fully-Qualified Names

Although rule names are unique within a grammar, separate grammars may use the same rule name. A later section introduces the `import` statement, which allows one grammar to reference rules from another grammar. When two grammars use the same rule name, a reference to that rule name may be ambiguous. *Qualified names* and *fully-qualified names* are used to resolve these ambiguities.

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

```
<COM.sun.greetings.english.hello>

<COM.sun.greetings.deutsch.gutenTag>
```

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

```
<english.hello>

<deutsch.gutenTag>
```

3. The `isWhitespace` method of the `java.lang.Character` class can be used to test for whitespace characters in the Unicode character set.

The following conditions apply to the use of rule names:

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

2.2.2 Resolving Names

It is an error to use an ambiguous reference to a rule name. 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. 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 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 rule name is never ambiguous.

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 or in whitespace-separated sequences. For example,

```
hello

konnichiwa

this is a test

open the directory
```

In Java Speech Grammar Format, a token is a Unicode 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.

Most recognizers operate *mono-lingually*, that is, listening for one language at any given time. Similarly, their vocabulary will normally contain the words of one language. Therefore, grammar definitions only contain tokens from a single language and the interpretation of tokens is language specific.

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 two possibilities:

- An application 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.

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 search for each space-separated token within the quotes.

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"), hyphens ("new-age"), and periods ("Mr.").

There are, however, many textual forms that are difficult for a recognizer to handle unambiguously. In these instances, a grammar developer should provide 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.”
- 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, rule names, tags and weights.

In this version there is no special meaning to comments beginning with `/**`, unlike Java code in which this form signifies a documentation comment.

3.0 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 declares the name of the grammar and may import rules from other grammars. The body defines the rules of the grammar, some of which may be public.

3.1 Grammar Name Declaration

The grammar's name must be declared as the first statement of that grammar. The format is either of the following:

```
grammar packageName.grammarName;

grammar grammarName;
```

The naming of packages and grammars is described in the section on *Grammar Names and Package Names* on page 2.

For example:

```
grammar COM.Sun.speech.apps.numbers;

grammar EDU.unsw.med.people;

grammar examples;
```

3.2 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 import statement imports a single rule referenced by its fully-qualified rule name (the rule named `<1stTo31st>` from the grammar named `COM.Sun.speech.app.index`).

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.

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

The name resolving behavior is defined earlier in this document in *Resolving Names* on page 5.

Note that even when an imported rule is referenced by its fully-qualified name, the corresponding import statement for the grammar is required. (This differs from the Java Programming Language.)

4.0 Grammar Body

The grammar body defines *rules*. The rule's definition is a logical combination of text that may be spoken, referred to as *tokens* or *terminals*, and *references* to other rules. A rule is defined once, and only once, in a grammar. The order of definition of rules is not significant. (These properties differ from some linguistic grammar formats.)

4.1 Rule Definitions

The patterns for defining a rule are:

```
<ruleName> = ruleDefinition ;

public <ruleName> = ruleDefinition ;
```

The rule name to be defined is followed by the equals sign, '=', a rule definition, and the closing semi-colon, ';'.

The simplest rule definitions are *references* to another rule or to a token. A reference to another rule uses its rule name, its qualified name, or its fully-qualified name. The following are examples defining the rule <x> by a simple rule reference, by a fully-qualified rule reference, and as a token:

```
<x> = <y>;

<x> = <COM.acme.grammar.y>;

<x> = word;
```

As explained below, more complex rule definitions are built by the following mechanisms:

- *Composition* of rule definitions as sequences of sub-rule definitions and sets of alternative sub-rule definitions

- *Grouping* using parentheses and brackets
- *Unary operators* for repetition of rule definitions
- Attachment of application-specific *tags*

4.1.1 Public Rules

Any rule in a grammar may be declared as public. A public rule is defined with the pattern:

```
public <ruleName> = ruleDefinition ;
```

A public rule has three possible uses:

- It can be *imported* into another grammar so that it may be referenced in the rule definitions of that grammar.
- It can be used as an *active* rule for recognition (i.e., a rule used by a recognizer to determine what may be spoken).
- It can be referenced locally.

Without the public declaration, a rule is implicitly private and so can only be referenced locally (in the grammar in which it is defined). Note that unlike the Java Programming Language, the Java Speech Grammar Format does not have keywords for `private` or `protected`.

4.2 Composition

4.2.1 Sequence

A rule definition can be a sequence of sub-rule definitions separated by white space. For example:

```
<where> = I live in Boston;

<statement> = this <object> is <OK>;
```

Each entry in the sequence must be spoken in order for the complete sequence to be spoken. 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 speakable tokens with references to other 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 rule definition. This includes the structures described below for alternatives, groups and so on. The items are separated by whitespace characters.

4.2.2 Alternatives

A rule definition can be a *set of alternative* rule definitions separated by '|' symbols (vertical bar) 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.

4.2.3 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 surrounded by forward slashes, e.g. /3.14/. The higher the weight, the more likely 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);
```

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 entry in a set of alternatives, then a weight must be specified for every alternative (the "all or nothing rule").
- Weights are floating point numbers that could be passed to the `java.lang.Float.valueOf(String)` method. 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 entry can never be spoken, as if the entry was not included in the alternatives list. (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.

4.3 Grouping

4.3.1 (Parentheses)

Any rule definition 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:

```
<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 each from the three sets of alternatives: for example, "open windows immediately" or "close doors later".

If a grouping surrounds a single definition, then the entity is defined to be a sequence of one item (not an alternative with only one option). For example:

```
(start)

( <end> )
```

4.3.2 [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".

4.4 Unary Operators

There are three *unary operators* in the Java Speech Grammar Format. The unary operators share the following features:

- They attach to the immediately preceding rule definition
- They have high precedence
- Only one unary operator can be attached to any rule definition

*4.4.1 * Kleene Star*

A rule definition followed by the asterisk symbol indicates that the immediate preceding rule definition 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: "don't crash".

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

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

matches "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".

4.4.2 + Plus Operator

A rule definition followed by the plus symbol indicates the rule definition 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 don’t crash,” but “don’t crash” is not legal.

The precedence of + is the same as *.

4.5 Tags

Tags provide a mechanism for grammar writers to attach application-specific information to rule definitions. These tag attachments do not affect recognition of a grammar. Instead, the tags are attached to the result object returned by the recognizer to the application. Applications may use these tags to simplify or enhance the processing of recognition results.

A tag may be attached to any part of a rule definition: to a token, to a rule reference, to a sequence or to a set of alternatives. When attaching to a sequence or set of alternatives, parentheses ‘()’ should be used to enclose the item being tagged.

The tag is a string delimited by curly braces ‘{}’. The tag must immediately follow the item being tagged (whitespace is allowed). For example:

```
<rule> = <action>{tag in here};

<command>= please (open {OPEN} | close {CLOSE}) the file;

<country> = Australia | 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};
```

Also, since only one unary operator may be attached to a rule definition (and a tag is a unary operator), the following are not permitted:

```
<badRule> = <action> {tag1} {tag2};

<badRule> = <action> * {tag2};

<badRule> = <action> {tag1} +;
```

With the addition of parentheses, the following is legal:

```
<OKrule> = ( <action> * ) {tag};
```

4.5.1 Using Tags

Tags 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 four following examples are rule definitions from four grammars used for four separate languages. The application would load the grammar for the language spoken by the user, but the tags would remain the same across all languages and thus simplify the application software that processes the results.

In the English grammar:

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

In the Japanese grammar:

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

In the German grammar:

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

In the French grammar:

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

4.6 Precedence

The following list defines the relative precedence of syntactic components of the Java Speech Grammar Format. The list is from highest to lowest precedence:

1. A string started with the quote symbol continues until the matching quote symbol.
2. '()' parentheses for grouping and '[]' for optional grouping.
3. The unary operators ('+', '*') and tag attachments apply to the tightest item immediately preceding them. To apply them to a sequence or to alternatives, use '()' or '[]' grouping.
4. Sequences of rule definitions.
5. '|' separated set of alternatives.

4.7 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 Java Speech 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 grammar definitions. However, this is strongly discouraged, as it introduces unnecessary complexity and potential maintenance problems.

Any right recursive rule can be re-written using unary operators, specifically Kleene star ‘*’ and the plus operator ‘+’. For example, the following rule definitions are equivalent:

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

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

Although re-write using ‘+’ and ‘*’ is equivalent, right recursive grammars are permitted because they allow simpler and more elegant representations of some grammars. Other forms of recursion (left recursion, embedded recursion) are not supported because this type of re-write cannot be guaranteed.

5.0 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.

```
/*** Header ***/
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.

```
/*** Header ***/
grammar COM.acme.commands;
import <COM.acme.politeness.startPolite>;
import <COM.acme.politeness.endPolite>;

/*** Body ***/
// e.g. “please move the window”
public <basicCommand> = <startPolite>* <command>
       [<endPolite>];

// e.g. “open the file”
<command> = <action> [the | a] <object>;
<action> = /10/ open |/2/ close |/1/ delete |/1/ move;
<object> = 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.

```
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>;
```