



Semantic Interpretation for Speech Recognition

W3C Working Draft 16 November 2001

This version:

<http://www.w3.org/TR/2001/WD-semantic-interpretation-20011116/>

Latest version:

<http://www.w3.org/TR/semantic-interpretation/>

Previous version:

(first publication)

Editors:

Luc Van Tichelen, Lernout & Hauspie Speech Products

Copyright ©2001 W3C® (MIT, INRIA, Keio), All Rights Reserved. W3C [liability](#), [trademark](#), [document use](#) and [software licensing](#) rules apply.

Abstract

This document defines the process of Semantic Interpretation for Speech Recognition and the syntax and semantics of semantic interpretation tags that can be added to speech recognition grammars to compute information to return to an application on the basis of rules and tokens that were matched by the speech recognizer. In particular, it defines the syntax and semantics of the contents of [Tags](#) in the [Speech Recognition Grammar Specification](#).

Semantic Interpretation may be useful in combination with other specifications, such as the [Stochastic Language Models \(N-Gram\) Specification](#), but their use with N-grams has not yet been studied.

Although the results of semantic interpretation are describing the meaning of a natural language utterance, the current specification does not specifically generate such information in the [Natural Language Semantics Markup Language for the Speech Interface Framework](#). It is believed that semantic interpretation can produce information that can be encoded in the NL Semantics Markup Language, but this is not ensured or enforced.

Status of this document

This document is a public W3C Working Draft for review by W3C members and other interested parties. It is a draft document and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use W3C Working Drafts as reference material or to cite them as other than "work in progress". A list of current public W3C Working Drafts can be found at <http://www.w3.org/TR>.

This specification describes the syntax and semantics for semantic interpretation tags in speech recognition grammars, and forms part of the proposals for the W3C Speech Interface Framework. It is intended to be used with Speech Recognition grammars as defined in [Speech Recognition Grammar Specification](#).

This document has been produced as part of the [W3C Voice Browser Activity](#), following the procedures set out for the [W3C Process](#). The authors of this document are members of the [Voice Browser Working Group \(W3C Members only\)](#).

This document is for public review, and comments and discussion are welcomed on the public mailing list

This version is outdated!

For the latest version, please look at <https://www.w3.org/TR/semantic-interpretation/>.

▲ expand

subscribe in the subject line (include the word *unsubscribe* if you want to unsubscribe). The [archive](#) for the list is accessible online.

Table of contents

1. [Introduction](#)
 - 1.1. [Semantic Results](#)
 - 1.2. [Basic Principles](#)
 - 1.3. [Why not simply use ECMAScript?](#)
2. [Normative References and Conformance](#)
 - 2.1. [Normative References](#)
 - 2.2. [Conformance](#)
 - 2.2.1. [Relation with ECMA-262](#)
 - 2.2.2. [Conforming Semantic Interpretation Tags](#)
 - 2.2.3. [Conforming Grammar Fragments and Grammar Documents with SI Tags](#)
 - 2.2.4. [Conformance Requirements for Conforming Semantic Interpretation Processors](#)
 - 2.2.5. [Conforming ABNF and XML Grammar Processors](#)
 - 2.3. [Notational Conventions](#)
3. [Expressions in Semantic Interpretation Tags](#)
 - 3.1. [Attributes and Semantic Values](#)
 - 3.2. [Semantic Interpretation Tags](#)
 - 3.3. [Syntax of Semantic Interpretation Tags](#)
 - 3.3.1. [Assignment Lists](#)
 - 3.3.2. [Assignment and Simple Assignment](#)
 - 3.4. [Syntax for semantic attributes](#)
 - 3.4.1. [Syntax for attributes of Grammar Rule Names \(GRN\)](#)
 - 3.4.2. [Syntax for attributes of Grammar Rule References \(GRR\)](#)
4. [Semantic Values and Types](#)
 - 4.1. [The Undefined Type](#)
 - 4.2. [The Null Type](#)
 - 4.3. [The Boolean Type](#)
 - 4.4. [The String Type](#)
 - 4.5. [The Number Type](#)
 - 4.6. [The Struct Type](#)
 - 4.7. [The List Type](#)
 - 4.8. [The Reference Type](#)
 - 4.9. [Type Conversion](#)
 - 4.9.1. [ToPrimitive](#)
 - 4.9.2. [ToBoolean](#)
 - 4.9.3. [ToNumber](#)
 - 4.9.4. [ToInteger](#)
 - 4.9.5. [ToString](#)
5. [Semantic Expressions](#)
 - 5.1. [Primary Expressions](#)
 - 5.1.1. [AttributeRef](#)
 - 5.1.2. [Literal](#)
 - 5.1.3. [List Literal](#)
 - 5.1.4. [Struct Literal](#)
 - 5.1.5. [The Grouping Operator](#)
 - 5.2. [Call Expressions](#)
 - 5.2.2. [Conversion to String](#)
 - 5.2.3. [Conversion to Boolean](#)
 - 5.2.4. [Conversion to Number](#)
 - 5.2.5. [Conversion to Integer](#)
 - 5.2.6. [Appending to List and Concatenating Lists](#)
 - 5.2.7. [Substring selection](#)
 - 5.3. [Unary Operators](#)
 - 5.4. [Multiplicative Operators](#)
 - 5.5. [Additive Operators](#)
 - 5.6. [Relational Operators](#)
 - 5.7. [Equality Operators](#)

- 5.8. [Binary Logical Operators](#)
 - 5.9. [Conditional Operator](#)
 - 5.10. [Semantic Expression](#)
 - 6. [Default Assignment](#)
 - 7. [Visibility Rules and order of tag evaluation for ABNF/XML Speech Recognition Grammar Format](#)
 - 7.1. [Logical Parse Structure](#)
 - 7.2. [Flat Parse List](#)
 - 7.3. [Visibility Rules, Scope](#)
 - 7.4. [Order of tag execution](#)
 - 7.5. [Examples](#)
 - 8. [Example Grammar with Semantic Interpretation Tags](#)
 - 9. [Future study](#)
 - 9.1 [Other shadow variables](#)
-

1. Introduction

This section is informative.

1.1. Semantic Results

Grammar Processors, and in particular speech recognizers, use a grammar that defines the words and sequences of words to define the input language that they can accept. The major task of a grammar processor consists of finding the sequence of words described by the grammar that (best) matches a given utterance, or to report that no such sequence exists.

In an application, knowing the sequence of words that were uttered is sometimes interesting but often not the most practical way of handling the information that is presented in the user utterance. What is needed is a computer processable representation of the information, the semantic result, more than a natural language transcript.

The Voice Browser working group is working on one such computer representation: [Natural Language Semantics Markup Language \(NLSML\)](#). It is expected that a later version of the Semantic Interpretation Tags specification will address how the results it computes can be represented with NLSML. The current version however doesn't specify how the results computed by semantic tags are eventually returned to the hosting environment.

When used with a VoiceXML Processor, it is expected that a Semantic Interpretation Tag Processor will convert the result into an ECMAScript object that can then be processed as specified in the VoiceXML 2.0 specification section 3.1.6 [Mapping Semantic Interpretation Results to VoiceXML forms](#).

Semantic Interpretation Tags provide a means to attach instructions for the computation of such semantic results to a speech recognition grammar.

This document defines the syntax and the semantics of Semantic Interpretation Tags for use with the [Speech Recognition Grammar Specification](#).

It is possible that Semantic Interpretation Tags as defined here can be used also with the [N-Gram Specification](#), but the current specification does not specifically address such use and does not guarantee that the Semantic Interpretation Tags as defined here are meeting the needs of such use.

1.2. Basic Principles

The basic principles for the Semantic Interpretation mechanism defined in this specification are the following:

- semantic information is represented as values associated with non-terminals
- expressions in Semantic Interpretation Tags are restricted to assignment statements
- the expression in the assignments is a proper subset of ECMAScript expressions, with some additional call expressions

- expression evaluation order is connected to the grammar rule definitions and the sequence of words in the utterance

1.3. Why not simply use ECMAScript?

While there was no explicit requirements document created for the properties of a semantic interpretation syntax, the working group gradually learned that there are some conflicting desires to be met.

Certainly, the Semantic Interpretation Tags must be easy to use by developers, and it should minimally provide the expressive power that is needed for the majority of applications. ECMAScript would meet these requirements.

On the other hand, there are concerns on performance and other implications from using ECMAScript (such as variable scoping, platform access etc.).

It was concluded that the semantic interpretation should be specified in such a way that

- there are no constructs that give rise to performance or side effects
- in what it supports it behaves as much as possible identical to ECMAScript so that developers familiar with ECMAScript find semantic interpretation intuitive to use
- the specification does not imply that an implementation of it requires a full ECMAScript interpreter or comes close to a full ECMAScript interpreter implementation
- if desired the specification can easily be implemented or emulated using an ECMAScript interpreter, with some features optimized or removed from it.

The current proposal presents a syntax that is both powerful enough but doesn't lead to performance concerns, and maximizes ease of use by defining the most complex part of the expressions as a proper subset of ECMAScript expressions.

2. Normative References and Conformance

2.1. Normative References

This document makes abundant references to the [ECMA-262 Standard "ECMAScript Language Specification", 3rd Edition - December 1999](#), further referenced as ECMA-262.

When a section of the ECMA-262 standard is referenced as normative in this document, that section with all its subsections are normative for this document. No other section of the ECMA-262 standard is normative for the Semantic Interpretation specification unless it is explicitly referenced as normative in this document. In particular, references to other sections in ECMA-262 out of sections referenced here, are not automatically normative for Semantic Interpretation.

For informative purposes, some text from the ECMA-262 has been copied in this document. Where that is done, unless otherwise specified, such text should be considered informative and the corresponding reference to the ECMA-262 standard is normative.

Future updates of ECMA-262 may have different section numbering and modified content in the sections referenced here. They do not become normative for this specification. In other words, references to ECMA-262 are always to the 3rd Edition - December 1999.

All sections in this specification are normative, unless otherwise indicated.

2.2. Conformance

2.2.1. Relation with ECMA-262

While this specification inherits a lot from the ECMA-262 standard, it does not require that a processor for this specification be built on an ECMAScript interpreter implementing ECMA-262.

There are many aspects of ECMA-262 that are not used or that are not relevant for the Semantic Interpretation specification. For example, Semantic Interpretation does not have objects nor depends on

objects for its implementation. Scoping in Semantic Interpretation is much simpler and quite different than in ECMA-262.

A few aspects of Semantic Interpretation are explicitly different from ECMA-262.

One of the most essential differences is that a Semantic Interpretation does not contain statements, but only assignments. Unlike ECMAScript, there are no blocks, no conditional statements, no iteration statements, no exception statements, no function definitions, no variable declarations.

The two forms of assignment in Semantic Interpretation are also incompatibly different from the assignment expression as in ECMA-262. For example, unlike ECMAScript assignments, a Semantic Interpretation assignment may be of the form "a.b.c = expression" where neither a nor a.b have yet been given a value. But the assignments contain a right-hand side expression that is a proper subset of ECMA-262 expressions plus a few additional call expressions.

Semantic Interpretation has the same primitive data types as ECMA-262 (Undefined, Null, Boolean, Number, and String), but lacks the Object data type. Instead, it has two complex data types (Struct and List).

As a consequence of these differences, it is not possible to take the content of a Semantic Interpretation as is and have it processed as ECMAScript.

But the intent of the specification is that it is possible to correctly and completely evaluate the expressions contained in the assignments using an unmodified ECMAScript interpreter, provided that the appropriate initialization, cleanup and data retrieval is done on the interpreter before and after processing the expression by some additional piece of software.

Note: there is some concern in the group that the piece of software mentioned here could be non-trivial. The group is inviting comments and is seeking feedback from implementation experience.

2.2.2. Conforming Semantic Interpretation Tags

A Semantic Interpretation Tag (SI Tag) is a conforming SI Tag if it is matching the syntax as defined in the normative sections in this document and the sections in ECMA-262 that are normatively referenced as defined in section 2.1.

There is no normative restriction on the size of a SI Tag.

2.2.3. Conforming Grammar Fragments and Grammar Documents with SI Tags

A stand-alone ABNF or XML Grammar Document or an XML Grammar Fragment with SI Tags is conforming if

- the document or grammar fragment is a conforming ABNF or XML document or fragment as defined by [the conformance requirements in the Speech Recognition Grammar Specification](#), and
- every tag in the grammar document or fragment is a conforming SI tag
- the [tag-format](#) for the grammar fragment or document is "semantics/1.0".

Informative

The Speech Recognition Grammar Specification provides a tag-format declaration that identifies the format of the contents of the tag element in a speech grammar. The tag-format to reference Semantic Interpretation Tags conforming with the present specification is defined here as "semantics/1.0". Note that this is the default tag-format in the current Speech Recognition Grammar Specification when no explicit tag-format is specified.

It is expected that future revisions of this specification will use higher version numbers.

Other tag-formats can be used with Speech Recognition Grammars; in this case the tag-format must be explicitly declared and must not be "semantics/x.y" (where x and y are any digits).

2.2.4. Conforming Semantic Interpretation Processors

A Semantic Interpretation Processor is a program that can parse and process Semantic Interpretation Tags to produce semantic values. Semantic Interpretation Processors are executed in a hosting environment (e.g. a grammar processor or VoiceXML processor).

A Conforming Processor

- Must be capable of accepting and processing any conforming Semantic Interpretation Tag as defined in 2.2.2.
- Must execute every conforming Semantic Interpretation Tag according to the normative sections in this specification.
- Should inform the hosting environment at the time it evaluates a conforming Semantic Interpretation Tag that causes a runtime error
- Must inform the hosting environment when it encounters a non-conforming Semantic Interpretation Tag that it can not process. A processor is free to inform the hosting environment of such non-conforming tag any time between loading the non-conforming SI Tag and evaluating the offending language construct in the non-conforming SI Tag. There is no requirement for a processor to process documents containing a non-conforming tag.
- May inform the hosting environment when it encounters a non-conforming SI Tag (even if it can process it as described below)
- May process a non-conforming SI Tag with a behavior that is defined in the processor's documentation. This allows conformant processors to handle proprietary extensions or tags with non-standard semantic tags, as long as the behavior for every conforming SI Tag in the document is not deviating from this specification.

Note: the group is considering adding a "strict" mode in which a semantic processor MUST inform the hosting environment when it encounters a non-conforming SI Tag, even if it can process it.

Informative

We anticipate that following will be the non-conforming conditions a processor may encounter:

1. Non-conforming document by developer error (or error in automatic document generation).
2. Not conforming by use of a proprietary semantic interpretation syntax in the grammar tags.
3. Not conforming by use of proprietary extensions to SI Tags.

The W3C Voice Browser Working Group has applied to IETF to register MIME types for both the ABNF and XML grammar forms (See Appendix G. [Media Types and File Suffix](#) of the Speech Recognition Grammar Specification)

The ABNF MIME type will identify ABNF grammars containing only conforming SI Tags. If the grammar contains tags of any other format then a different MIME type must be used.

Similarly, the XML grammar MIME type will identify XML grammars containing only conforming SI Tags. If the grammar contains tags of any other format then a different MIME type must be used.

A grammar that contains tags in a format other than conforming SI Tags must have an explicit tag format declaration specifying the format (see Speech Recognition Grammar Specification 4.1.5 [Tag Format Declaration](#)). The tag-format for a grammar that contains conforming Semantic Interpretation Tags is "semantics/1.0".

Note: a VoiceXML 2.0 processor will require support for Semantic Interpretation Tags as defined here, but will allow to support other grammar formats or SRGF with other tags in addition (probably identified by other MIME type).

2.2.5. Conforming ABNF and XML Grammar Processors

An ABNF or XML Grammar Processor is a conforming processor if

- it is a conforming ABNF or XML Grammar Processor as defined in the Speech Recognition Grammar Specification, and

- it is a conforming Semantic Interpretation Processor

2.3. Notational Conventions

Throughout the specification following abbreviations will be used:

Abbreviation	Description
GRN	Grammar Rule Name, the rule name that is in the left-hand side of a speech grammar rule definition.
GRR	Grammar Rule Reference, a rule reference in the rule expansion that is in the right-hand side of a speech grammar rule definition.
ES <i>n</i>	Shorthand notation for ECMA-262 Section number <i>n</i>
SI Tag	Semantic Interpretation Tag

This specification uses the notational conventions for Syntactic and Lexical Grammars as given in ES 5.1, and the same Algorithm Conventions as in ES 5.2.

3. Expressions in Semantic Interpretation Tags

3.1. Attributes and Semantic Values

Semantic Interpretation Tags compute semantic values. During the semantic interpretation processing, semantic values are stored in attributes that are associated with the rules in the grammar.

Every rule always has exactly one attribute that holds a semantic value. The attribute is not named, but is referred to by referring to the name of the rule as defined in section 3.4.1.

During semantic interpretation evaluation, attribute values of a rule that is referenced (Grammar Rule Reference, or GRR) can be evaluated to compute the value for the attribute of the rule that is named in the grammar rule definition (Grammar Rule Name, or GRN).

Attributes can hold semantic values of the allowed semantic types defined in section 4. They are not explicitly typed. Attributes that have not been assigned a value are of type Undefined. While many rules will compute attributes of string or numeric type, it is expected that most root rules will compute more complex types such as structs.

Attributes are not objects, and there is no object type in the semantic types.

Next to the attribute, every GRR also has an associated text variable of type string, which holds the substring (series of tokens) in the utterance that is governed by that GRR. Text variables are not part of the GRR's attribute and can not be modified.

The semantic result for an utterance is the value of the attribute of the root rule when all semantic interpretation evaluations have been completed. It is outside the scope of this specification to define how the semantic result is communicated to the application.

Informative Note:

In the context of the W3C voice browser architecture, the semantic result will either be transformed into the Natural Language Semantics mark-up (more precisely, in the interpretation element), or directly be cast into ECMAScript variables in the VoiceXML interpreter (see VoiceXML2.0 section 3.1.6. [Mapping Semantic Interpretation Results to VoiceXML forms](#)).

3.2. Semantic Interpretation Tags

A Semantic Interpretation Tag can either hold a series of assignments, or a simple assignment, or it can be empty.

```
ABNFSemanticTag ::
    AssignmentList
    SimpleAssignment
```

This specification assumes the Semantic Interpretation Tag to consist of text characters, with all escapes, delimiters and other mark-up added by the language in which the Semantic Interpretation Tag is used, removed or resolved before being processed as a Semantic Interpretation Tag.

Informative:

Semantic Interpretation Tags are added in the string content of tags in the grammar rule expansion, as described in [Section 2.6 Tags](#) of the Speech Recognition Grammar Specification. This specification further uses the term Semantic Interpretation Tag (or SI Tag) to refer to such tag.

Below are two example formats of SI Tags in the Speech Recognition Grammar Specification.

ABNF Form

In the ABNF grammar format, SI Tags are enclosed in curly braces.

```
ABNFSemanticTag :
{
{ AssignmentList }
{ SimpleAssignment }
```

XML Form

In the XML grammar format, SI Tags are specified as the content of the tag element.

```
XMLSemanticTag:
<tag/>
<tag> </tag>
<tag> AssignmentList </tag>
<tag> SimpleAssignment </tag>
```

3.3. Syntax of Semantic Interpretation Tags

3.3.1. Assignment Lists

An Assignment List holds zero, one or more assignments, separated by semicolon. The semicolon can also be optionally added anywhere around assignments.

```
AssignmentList :
;
AssignmentList ;
; AssignmentList
Assignment
AssignmentList ; Assignment
```

The production *AssignmentList : Assignment* is evaluated as follows:

1. Evaluate Assignment .

The production *AssignmentList : AssignmentList ; Assignment* is evaluated as follows:

1. Evaluate AssignmentList .
2. Evaluate Assignment .

The AssignmentList is not defined by reference to ECMA-262. It resembles StatementList in ES12.1, but differs from a StatementList in at least the following ways:

- there is no *Statement*, but *Assignment*
- there is no exception handling. Since there are no throw statements in Semantic Interpretation, the only difference here is about run-time errors. These are handled in Semantic Interpretation as defined in section 2.2.4.
- Assignments and AssignmentLists do not return values (a value wouldn't be useful since the Assignment can not be used in an expression).

Note: ECMA-262 defines a normalization of the input to automatically insert semicolons where possible. That is more unrestricted than what is required according to the AssignmentList definition above (there must be a semicolon between assignments, and there may be more). The use of semicolons in the AssignmentList definition is a legal subset of what is accepted as valid by ECMA-262.

3.3.2. Assignment and Simple Assignment

Informative:

In an assignment, the value of the right-hand side expression is computed as described in section 5, and assigned to the attribute or property stated in the left-hand side of the assignment.

Assignments can only be done to the attribute or to a property of the attribute of a GRN, not on attributes or properties of GRRs. Assignments can not modify text variables.

If the assignment is to the attribute of the GRN (and not to a property of it), the assignment can be stated using a simplified syntax (Simple Assignment) that only states the right-hand side expression, with the effect of assigning the result of that expression to the attribute of the GRN. Simple Assignments can not be used in a list of assignments inside a SI Tag, but there can be more than one SI Tag with a simple assignment in any grammar rule expansion.

While the assignment behaves much like an assignment operator in ECMAScript, the assignment is not defined by reference to ECMA-262. This is indeed one of the few places where the Semantic Interpretation Tags are different from ECMAScript:

- Assignments are not themselves expressions. They don't return a result, and can't be used in an expression in another assignment.
- The expression at the left-hand side is restricted to identifiers (see section 4)
- The expression at the right-hand side is restricted to a subset of ECMA-262 expressions (see section 5)

Notwithstanding these differences,

- Assignments work by reference as in ECMAScript
- Left-hand side identifiers are a subset of ECMA-262 LeftHandSideExpressions
- Right-hand side expressions are a subset of ECMA-262 expressions plus some additional call expressions

As a consequence, assignments follow an exact subset of ECMA-262 syntax, and have semantics that is intuitive for users who are familiar with ECMAScript.

Normative:

```
Assignment :
    WriteablePropertyRef = SemanticExpression

SimpleAssignment :
    SemanticExpression
```

The production *Assignment* : *WriteablePropertyRef* = *SemanticExpression* is evaluated as follows:

1. Evaluate *WriteablePropertyRef* .
2. Evaluate *SemanticExpression* .
3. Call *GetValue*(*Result*(2)).
4. Call *PutValue*(*Result*(1), *Result*(3)).

The production *SimpleAssignment* : *SemanticExpression* is evaluated as follows:

1. Evaluate a reference to the GRN attribute .
2. Evaluate *SemanticExpression* .
3. Call *GetValue*(*Result*(2)).
4. Call *PutValue*(*Result*(1), *Result*(3)).

The Assignment and SimpleAssignment are not defined by reference to the ECMA-262 standard. They differ from the assignment operators defined in section 11.13 of ECMA-262 in at least the following aspects:

- there is no LeftHandSideExpression, but a WriteablePropertyRef
- the WriteablePropertyRef is optional
- there is no AssignmentExpression, but a SemanticExpression
- an Assignment or SimpleAssignment is not itself a SemanticExpression
- Assignment and SimpleAssignment do not return a value
- there is no compound assignment
- conditional expressions are not part of Assignment or SimpleAssignment (they are part of SemanticExpression)

3.4. Syntax for semantic attributes

3.4.1. Syntax for attributes of Grammar Rule Names (GRN)

Informative

The GRN's attribute can both be evaluated and assigned to.

It is identified by a dollar sign \$.

If the GRN attribute is of type struct, struct properties can be individually accessed by the property name or by \$.propertyname

WriteablePropertyRef has the appearance of ECMAScript member expressions. It is defined in terms of ECMAScript Identifiers, but it is not the same as an ECMA-262 MemberExpression or LeftHandSideExpression. For example, WriteablePropertyRef doesn't have a new operator nor call expressions or array member expressions.

Examples

```
$      identifies the GRN attribute
pizza  identifies the pizza property of the GRN attribute of type struct
$.pizza identifies the pizza property of the GRN attribute of type struct
a.b    identifies the property b of the struct a
       that is itself a property of the GRN attribute of type struct
```

As in ECMAScript, a WriteablePropertyRef can not be a reserved word nor can literals in its name be a reserved word. A WriteablePropertyRef can also not be a ReadOnlyPropertyRef, but parts of it can.

Examples

```
for      illegal: for is a reserved word
$.for    illegal reference to the for property of the GRN attribute

$a       illegal: $a is ReadOnlyPropertyRef
$.a      legal reference to the $a property of the GRN attribute
```

Issue: Excluding ReadOnlyPropertyRef makes it a syntax error to attempt assignment to a GRR or to the GRN or GRR text variable. The group also considered to rather accept ReadOnlyPropertyRef as valid syntax but ignore it, similar to attempts to assigning to read-only object properties in ECMA-262 simply being ignored. Note that excluding ReadOnlyPropertyRef here makes ReadOnlyPropertyRef and WriteablePropertyRef mutually exclusive, which is important for disambiguation in the AttributeRef production (3.4.2). The group is interested in reviewer feedback on this issue.

Normative

```
WriteablePropertyRef ::
  Identifier
  WriteablePropertyRef . Identifier
but not ReadOnlyPropertyRef
```

```

SIReservedWord ::
    ECMAReservedWord
    CallIdentifier

Identifier ::
    is defined by reference to ECMA-262 Section 7.6, but using SIReservedWord
wherever ReservedWord is used.

ECMAReservedWord ::
    is defined by reference to the definition of ReservedWord in ECMA-262 Section 7.5

```

The production *WritablePropertyRef* :: *Identifier* is evaluated as follows:

1. If Identifier is a dollar sign \$, return a value of type Reference to the GRN attribute
2. If the GRN attribute is not of type struct, make a new struct with no properties and assign it to the GRN attribute, overwriting its current contents.
3. If the GRN attribute's struct doesn't contain a property whose name matches the characters in Identifier, create a property on this struct with as name the characters in Identifier.
4. Return a value of type Reference to the property of the GRN attribute's struct whose name matches the characters in Identifier

The production *WritablePropertyRef* :: *WritablePropertyRef* . *Identifier* is evaluated as follows:

1. Evaluate *WritablePropertyRef* .
2. If Type(GetValue(Result(1))) is not struct, make a new struct with no properties and assign it to Result(1), overwriting its current contents.
3. If the struct referenced by Result(1) doesn't contain a property whose name matches the characters in Identifier, create a property on Result(1) with as name the characters in Identifier.
4. Return a value of type Reference to the property of the struct referenced by Result(1) whose name matches the characters in Identifier

WritablePropertyRef is not defined by reference to ECMA-262, but the *Identifier* and *ReservedWord* used in its expansion are defined by reference to ECMA-262.

WritablePropertyRef differs from ECMA-262 in at least the following aspects:

- it doesn't have call expressions, new expressions, function expressions
- it doesn't have all types of primary expressions, but only Identifiers
- identifiers are not evaluated as defined in ECMA-262 section 10.1.4.
- reference to a property that doesn't exist automatically creates a parent struct with such property (no need for new operator)

3.4.2. Syntax for attributes of Grammar Rule References (GRR)

Informative

The GRR's attributes, properties and text variables can only be evaluated, not assigned to.

A GRR's attribute is identified by *\$RuleName*, where *RuleName* is the name of the GRR.

If the GRR attribute is of type struct, individual properties can be identified by *\$RuleName.Identifier*, where *RuleName* is the name of the GRR and *Identifier* is the name of the property.

A GRR's text variable is identified by *\$RuleName\$.text*, where *RuleName* is the name of the GRR.

A GRR's attribute, properties and text variable can only be referenced in SI Tags that appear after (to the right or below) the GRR in the grammar expansion, and only if the GRR was used in the expansion that matched the input utterance. See visibility rules in [section 7](#) for a more detailed description of when GRR can be referenced in SI Tags, using the concept of the logical parse structure and the flat parse list.

A GRR can also be referenced through *\$\$*, which denotes the last GRR that was used in the expansion matching the utterance but still before to the tag (i.e. the last element of type rule reference before the tag in the flat parse, see [section 7.2](#)).

In an expression, both the GRR's and GRN's attributes and properties can be evaluated. The notation of the GRN's attribute and properties is defined in 3.4.1; the syntax for referring to the GRN's attribute and properties is the same as WriteablePropertyRef but the production is different and defined below as WriteablePropertyRefName.

Special Rules (NULL, VOID, GARBAGE) can not be evaluated.

AttributeRef has the appearance of ECMAScript member expressions. It is defined in terms of ECMAScript Identifiers, but it is not the same as an ECMA-262 MemberExpression or LeftHandSideExpression. For example, *AttributeRef* doesn't have a new operator nor call expressions or array member expressions.

Examples

\$	the attribute of the GRN
\$.prop	the property prop of the GRN's attribute
prop	the property prop of the GRN's attribute
\$rname	the attribute of GRR \$rname
\$rname.prop	the property prop of the attribute of GRR \$rname
\$rname\$.text	the text variable of the GRR \$rname
\$\$	\$\$ is shorthand for the closest matching GRR before the SI Tag
\$\$\$.prop	Property prop of closest matching GRR before the SI Tag
\$\$\$\$.text	the text variable of the closest matching GRR before the SI Tag

Issue: The group has some concern that the use of the dollar sign is overloaded, and is looking for suggestions that can make this syntax both intuitive, connected with the grammar specification and not conflicting with XML and ES identifier syntax.

Normative

AttributeRef is not defined in ECMA-262, but the syntax of AttributeRef is a subset of the syntax of the ECMA-262 Identifier. Evaluation of AttributeRef does not follow the ECMA-262 Identifier scoping rules, but the rules defined below.

```

AttributeRef ::
    WriteablePropertyRefName
    ReadOnlyPropertyRef

WriteablePropertyRefName ::
    Identifier
    WriteablePropertyRefName . Identifier
but not ReadOnlyPropertyRef

ReadOnlyPropertyRef ::
    RuleRefAttribute
    RuleRef $. Specialname
but not SIReservedWord

RuleRefAttribute:
    RuleRef
    RuleRefAttribute . Identifier

RuleRef ::
    $ Rulename
    $$

Specialname :: one of
    text

Rulename ::
    A rulename as defined in Section 3.1
    of the Speech Recognition Grammar Specification.

```

The productions *RuleRef :: \$ Rulename*, and *RuleRef :: \$\$* are evaluated as follows:

1. Create the flat parse list for the current rule application, as defined in [section 7.2](#).
2. Use result(1) and remove from it all elements after the SI Tag element currently being evaluated
3. Use result(2) and remove all token and tag elements from it

4. If *RuleRef* is not a double dollar sign \$\$, use result(3) and remove all references to GRRs that have a name different than *\$RuleName* ; else use result(3)
5. If result(4) is an empty list, throw a ReferenceError exception.
6. Return a value of type Reference to the last GRR referenced in result(4).

The production *RuleRefAttribute* :: *RuleRef* is evaluated as follows:

1. Evaluate *RuleRef*
2. return a value of type Reference to the attribute of the GRR referenced in result(1)

The production *RuleRefAttribute* :: *RuleRefAttribute* . *identifier* is evaluated as follows:

1. Evaluate *RuleRefAttribute*
2. If Type(GetValue(Result(1))) is not Struct, throw a ReferenceError exception.
3. If the struct referenced by result(1) doesn't contain a property who's name matches the characters in Identifier, return **undefined**
4. Return a value of type Reference to the property of the struct referenced by Result(1) who's name matches the characters in Identifier

The production *ReadOnlyPropertyRef* :: *RuleRefAttribute* is evaluated as follows:

1. Evaluate *RuleRefAttribute*
2. Return result(1)

The production *ReadOnlyPropertyRef* :: *RuleRef* \$.*text* is evaluated as follows:

1. Evaluate *RuleRef*
2. return a value of type Reference to the text variable of the GRR referenced in result(1)

The production *WriteablePropertyRefName* :: *Identifier* is evaluated as follows:

1. If Identifier is a dollar sign \$, return a value of type Reference to the GRN attribute
2. If the GRN attribute is not of type struct, throw a ReferenceError exception.
3. If the GRN attribute's struct doesn't contain a property who's name matches the characters in Identifier, return **undefined**
4. Return a value of type Reference to the property of the GRN attribute's struct who's name matches the characters in Identifier

The production *WriteablePropertyRefName* :: *WriteablePropertyRefName* . *Identifier* is evaluated as follows:

1. Evaluate *WriteablePropertyRefName* .
2. If Type(GetValue(Result(1))) is not struct, throw a ReferenceError exception.
3. If the struct referenced by Result(1) doesn't contain a property who's name matches the characters in Identifier, return **undefined**
4. Return a value of type Reference to the property of the struct referenced by Result(1) who's name matches the characters in Identifier

The production *AttributeRef* :: *WriteablePropertyRefName* and *AttributeRef* :: *ReadOnlyPropertyRef* are evaluated as follows:

1. Evaluate *WriteablePropertyRefName* or *ReadOnlyPropertyRef*
2. Return Result(1)

4. Semantic Values and Types

A Semantic Value is an entity that takes on one of eight types. There are eight types (Undefined, Null, Boolean, String, Number, Struct, List, and Reference). Unlike ECMA-262, there is no Object type, and as a consequence there are no Boolean Objects, Number Objects etc.

Values of type Reference are used only as intermediate results of expression evaluation and cannot be stored in Semantic Attributes.

The semantic types are divided in two categories:

- primitive types: Undefined, Null, Boolean, String, and Number
- complex types: Struct and List

4.1. The Undefined Type

The Undefined Type is defined as in ES 8.1, except that values are stored in attributes instead of variables:

The Undefined type has exactly one value, called **undefined**. Any attribute or attribute property that has not been assigned a value has the value **undefined**.

4.2. The Null Type

The Null Type is defined by reference to the Null Type definition in ES 8.2

4.3. The Boolean Type

The Boolean Type is defined by reference to the Boolean Type definition in ES 8.3

4.4. The String Type

The String Type is defined by reference to the String Type definition in ES 8.4

4.5. The Number Type

The Number Type is defined by reference to the Number Type definition in ES 8.5

4.6. The Struct Type

This section is normative but in the event that it would conflict with the normative behavior described in `WritablePropertyRef` and `ReadOnlyPropertyRef`, the latter sections take precedence over this section.

A struct is an unordered set of property-value pairs. A struct has at least one property.

Every property of a struct holds a value of any of the allowed semantic value types (Undefined, Null, Boolean, Number, String, List, Struct). This implies that structs can be nested.

A struct can be referenced as a whole, including all its properties. Struct properties can also be individually referenced for evaluation and assignment. See `WritablePropertyRef` and `ReadOnlyPropertyRef` for the syntax.

Structs are automatically created by referencing or assigning to a property of an attribute, or they can be created with a struct literal. This implies that a struct always has a property (because it is the property that makes a value a struct, and because struct literals always have at least one element). There is no explicit way to create a struct before assigning values to a property of it.

The reference to a struct is lost by assigning a simple type value to the struct instead of to one of its properties. There is no explicit way to delete structs.

Because of the definition of assignments in Section 3.3, assignment of a struct to an attribute or an attribute's property is done by reference and not by copy. This means the data is shared, and any subsequent modification of the struct will also be reflected in the attribute or property that holds a reference to it.

Note that a struct is not the same as an ECMAScript object even though the syntax resembles member expressions. Structs are unlike ECMAScript objects for example because they lack constructors, methods, prototypes etc.

Example

The Semantic Interpretation Tag

```
{a.b.c = 5; x = $.a; a.b = 7}
```

will compute as follows (assuming the GRN's attribute is currently undefined):

1. Evaluate `a.b.c= 5;`

This automatically creates a property a on the GRN attribute; makes property a a struct with property b; and makes b a struct with property c. C is assigned the number value 5.

After this evaluation, the value of the GRN attribute is:

```
$={
  a : {
    b: {
      c: 5
    }
  }
}
```

2. Evaluate x = \$.a;

This automatically creates an additional property x on the GRN attribute. The other properties are not destroyed.

The value of x is a reference to the property a of the GRN attribute.

After this evaluation, the value of the GRN attribute is:

```
$={
  a : {
    b: {
      c: 5
    }
  }
  x : {
    b: {
      c: 5
    }
  }
}
```

3. Evaluate a.b = 7

This assigns the value 7 to property b of struct a. Since b was a struct (with property c), that property c is automatically destroyed and b now becomes of number type.

This also modifies the values in struct x because x holds a reference to a.

After this evaluation, the value of the GRN attribute is:

```
$={
  a : {
    b: 7
  }
  x : {
    b: 7
  }
}
```

4.7. The List Type

A list is an ordered, dynamic set of elements each holding a value.

Every element of a list holds a value of any of the allowed semantic value types (Undefined, Null, Boolean, Number, String, List, Struct). This implies that lists can be nested.

A list is created by assigning a list literal to an attribute or a property of it. There is no explicit operator to create a list.

Lists can only grow. A number of utility functions are available that append or concatenate information to an existing list (see section 5.2.6). There is no mechanism to delete elements from a list.

The reference to a list is lost when a value of a different type is assigned to the attribute or property holding the list. There is no explicit operator to delete a list.

Values of individual list elements cannot be referenced; lists can only be referenced as a whole.

A list is not the same as a JavaScript array and is not an ECMAScript object. In particular, the utility functions for working with lists are not methods of the list but call expressions taking one or more lists as arguments (see section 5.2.6)

The List Type is also not defined as the List Type in ECMA-262 section 8.8, which only defines the List Type as an internal type. In Semantic Interpretation, the List Type is a language data type.

Lists resemble ECMAScript Arrays in the literal syntax (see 5.1.3). They are unlike ES Arrays because they lack constructors, methods, prototypes etc.

Examples

The following line creates a list containing 4 string elements, using a list literal:

```
toppings = ["onion", "mushrooms", "anchovies", "artichokes"];
```

The Append operator allows adding elements to a list:

```
toppings = Append(toppings, "mozzarella");
```

Lists can contain elements of different types, including other lists. Note the use of [] to create an empty list to add elements to:

```
order = Append([], "tea")
order = Append(order, topping);
order = Append(order, 2);
```

When the above 5 statements would be executed in sequence, order would hold the following list:

```
["tea", ["onion", "mushrooms", "anchovies", "artichokes", "mozzarella"], 2]
```

4.8. The Reference Type

The internal Reference Type is not a language data type. It is defined by this specification purely for expository purposes. The Reference Type is used to explain the behavior of the assignment, and the Append and Concat call operators.

A Reference is a reference to a Struct, to a property of a Struct, to a List, or to a String. A Reference consists of two components, the *base* and the *property name*.

The following abstract operations on references are used in this specification: GetBase, GetPropertyName, GetValue, and PutValue.

The Reference Type is not defined as in ES8.7. The Reference Type differs from the Reference Type in ECMA-262 in at least the following ways:

- References are not to Object properties, but to Structs and Lists

Despite these differences, the Reference Type serves the same expository purpose as in the ECMA-262 specification and yields assignment by reference behavior for the assignment statement.

Informative

The Reference can be thought of as a pointer to a Struct or to a property of a Struct (or a property of a property etc.), or to a List. When a semantic value is of type Struct, it really holds a reference value that points to a Struct. Similarly, a semantic value of type List really holds a reference value identifying a List.

This results in assignments of type Struct and List to operate by reference. The values of the Struct or List are not copied, only the reference to the Struct or List is copied.

This behavior is similar to assignments of Objects in ECMAScript.

4.8.1. GetBase(V)

GetBase(V) returns the base of the reference V (the Struct, List or String pointed to).

4.8.2. GetPropertyName(V)

GetPropertyName(V) returns the property name component of the reference V, or null for references to a String, a List or to an entire Struct.

4.8.3. GetValue(V)

GetValue(V) is evaluated as follows:

1. If Type(V) is not Reference, return V
2. Call GetBase(V)
3. If Result(2) is of type String, return Result(2)
4. If Result(2) is of type List, return Result(2)
5. Call GetPropertyName(V)
6. If Result(5) is null, return Result(2)
7. Return the value of the property of the Struct in Result(2) whose name matches Result(5)

GetValue is different from GetValue in ES8.7.1 in following ways:

- There can not be a reference error, as there are no references with base null
- References can be to entire String (for .text variable), List or Struct
- The value is not obtained through [[get]]

4.8.4. PutValue(V,W)

PutValue(V,W) is evaluated as follows:

1. Set the value of the property GetPropertyName(V) in the Struct GetBase(V) to W
2. Return

PutValue is different from PutValue in ES8.7.2 in following ways:

- There can not be a reference error, as PutValue is only used in assignments with WriteablePropertyRef as argument. Any WriteablePropertyRef always returns a value of type Reference to a Struct with a property name that exists on the struct.
- Setting the value is not done through [[put]]
- There is no way to set values on the global object as such object doesn't exist in semantic interpretation.

4.9. Type Conversion

Informative

Semantic Interpretation is a latently typed language, meaning that types are associated with values instead of properties. No type declaration is necessary in Semantic Interpretation Tags. However, most operators require specific data types as their operands. Semantic Interpretation supports automatic data type conversion to meet the requirements of these operators. The conversion rules are defined in this section.

The conversion rules are the same as in ECMA-262, but there are specific rules defined for conversions from types that are not available in ECMA-262 (List, Struct).

Normative

The Semantic Interpretation runtime system performs automatic type conversion as needed. To clarify the semantics of certain constructs it is useful to define a set of conversion operators. These operators are not a part of the language; they are defined here to aid the specification of the semantics of the language. The conversion operators are polymorphic; that is, they can accept a value of any standard type, but not of type Reference (the internal type).

The automatic conversions are defined in this section. Where the Semantic Interpretation specification references sections in ECMA-262, the type conversion operators in these referenced sections in ECMA-262 must be evaluated using the definitions of the type conversion operators in sections 4.9.1 to 4.9.5, instead of as described in ECMA-262 section 9, unless the contrary is explicitly mentioned.

4.9.1. ToPrimitive

The operator ToPrimitive takes a Value argument and an optional PreferredType argument. The operator ToPrimitive converts its value argument to a non-Object type. Conversion occurs according to the following table:

Input Type	Result
Undefined	The result equals the input argument (no conversion).
Null	The result equals the input argument (no conversion).
Boolean	The result equals the input argument (no conversion).
Number	The result equals the input argument (no conversion).
String	The result equals the input argument (no conversion).
Struct	The result is an empty string.
List	The result is an empty string.

Note that conversions of List to primitive are not resulting to the same effect as an Array object would have in ECMAScript. The conversions of Struct and List to primitive are the effect of conversion to primitive of an ECMAScript object that defines that the [[defaultvalue]] of the object is always the empty string.

Note that ToPrimitive doesn't use the optional PreferredType argument here (all conversions are indifferent from this argument), but we have kept the same signature from the operator as defined in ES 9.

4.9.2. ToBoolean

The operator ToBoolean converts its argument to a value of type Boolean according to the following table:

Input Type	Result
Undefined	false
Null	false
Boolean	The result equals the input argument (no conversion).
Number	The result is false if the argument is +0 , -0 , or NaN ; otherwise the result is true .
String	The result is false if the argument is the empty string (its length is zero); otherwise the result is true .
Struct	true
List	true

4.9.3. ToNumber

The operator ToNumber converts its argument to a value of type Number according to the following table:

Input Type	Result
Undefined	NaN
Null	+0
Boolean	The result is 1 if the argument is true . The result is +0 if the argument is false .
Number	The result equals the input argument (no conversion).
String	As defined in ECMA-262 section 9.3.1

Input Type	Result
Struct	Apply the following steps: 1. Call ToPrimitive(input argument, hint Number) 2. Call ToNumber(Result(1)) 3. Return Result(2)
List	Apply the following steps: 1. Call ToPrimitive(input argument, hint Number) 2. Call ToNumber(Result(1)) 3. Return Result(2)

Note that on List and Structs this definition with the definitions of ToPrimitive and ToString will always result in **NaN**.

4.9.4. ToInteger

The operator ToInteger is evaluated as described in ECMA-262 section 9.4. In this evaluation, the evaluation of ToNumber is done according to the definition in section 4.9.3 (and not directly as defined in ES 9.3)

4.9.5. ToString

The operator ToString converts its argument to a value of type String according to the following table:

Input Type	Result
Undefined	"undefined"
Null	"null"
Boolean	If the argument is true , then the result is " true ". If the argument is false , then the result is " false ".
Number	As defined in ECMA-262 section 9.8.1
String	Return the input argument (no conversion)
Struct	Apply the following steps: 1. Call ToPrimitive(input argument, hint String) 2. Call ToString(Result(1)) 3. Return Result(2)
List	Apply the following steps: 1. Call ToPrimitive(input argument, hint String) 2. Call ToString(Result(1)) 3. Return Result(2)

Note that ToString conversion on List is different from what ECMAScript does on Arrays. The ToString conversion of a Struct is not defined in ECMAScript (i.e. it is defined that objects can provide a [[defaultvalue]] method used by ToPrimitive but objects don't need to define a [[defaultvalue]] method).

5. Semantic Expressions

Informative

This section describes the semantic expressions that can appear in the right-hand side of an assignment or that constitute a simple assignment.

Semantic expressions are a subset of expressions in ECMA-262, with some additional call expressions that are not defined in ECMA-262.

Semantic expressions only involve constants, attribute references, and operator calls.

The expression sub-language is thus fully functional as it does not involve side-effecting constructs or operators.

5.1. Primary Expressions

Normative

```
PrimaryExpression:
  AttributeRef
  Literal
  ( SemanticExpression )
```

PrimaryExpression syntax is a subset of the ECMA-262 PrimaryExpression syntax (ES 11.1):

- no **this** keyword
- AttributeRef instead of Identifier (AttributeRef syntax is subset of Identifier syntax)
- no ObjectLiteral but StructLiteral as part of Literal (StructLiteral syntax is subset of ObjectLiteral syntax)
- no ArrayLiteral but ListLiteral as part of Literal (ListLiteral syntax is subset of ArrayLiteral syntax)
- SemanticExpression instead of Expression (SemanticExpression syntax is subset of Expression syntax)

5.1.1. AttributeRef

AttributeRef is evaluated as defined in section 3.4

5.1.2. Literal

```
Literal ::
  NullLiteral
  BooleanLiteral
  NumericLiteral
  StringLiteral
  ListLiteral
  StructLiteral

NullLiteral ::
  is defined by reference to ECMA-262 section 7.8.1

BooleanLiteral ::
  is defined by reference to ECMA-262 section 7.8.2

NumericLiteral ::
  is defined by reference to ECMA-262 section 7.8.3, and the reference
  it makes to ECMA-262 section 8.5

StringLiteral ::
  is defined by reference to ECMA-262 section 7.8.4, and the references
  it makes to ECMA-262 section 7.8.3, section 6, and the second sentence in section 2.
```

Informative

The definition of Literal is different from the definition in ECMA-262 section 7.8. In particular, it also contains ListLiteral and StructLiteral. In ECMA-262, the ObjectLiteral and ArrayLiteral are not Literal but Expression.

5.1.3. List Literal

A list literal is an expression describing a value of type array.

```
ListLiteral:
  [ ]
```

```

[ ListElements ]

ListElements:
  Literal
  ListElements , Literal

```

The production *ListLiteral* : [] is evaluated as follows:

1. Return a reference to a list with no elements

The production *ListLiteral* : [*ListElements*] is evaluated as follows:

1. Create a list with no elements
2. For each *Literal* in *ListElements* from left to right do:
 - 2.1. append a value to the list of the type and the value of the *Literal*
3. Return a reference to the list built in 2.

Informative

The ListLiteral syntax is a subset of the ArrayLiteral syntax in ECMA-262 section 11.1.4, with a somewhat different semantics. In particular:

- A ListLiteral contains only Literals, no expressions
- There is no elision in ListLiteral
- The resulting value is not an array but a reference to a list

Examples

Below are some examples using list literals. Note that these are primarily given for illustration purposes of list literals, not as examples of good style of using lists in semantic interpretation tags.

The following line creates a list containing 4 string elements, using a list literal:

```
toppings = ["onion", "mushrooms", "anchovies", "artichokes"];
```

List Literals can contain other list literals:

```
coordinates = [ [15, 23], [47, 45] ];
```

Lists can contain elements of mixed types:

```
phonebook = [ ["John", 9088], ["Jeff", undefined], ["Lisa", 7083] ];
```

5.1.4. Struct Literal

```

StructLiteral:
  { StructElements }

StructElements:
  Pair
  StructElements , Pair

Pair:
  PropertyName : Literal

PropertyName :
  Identifier

```

The production *StructLiteral* : { *StructElements* } is evaluated as follows:

1. Create a struct with no properties
2. For each *Pair* in *StructElements* from left to right do
 - 2.1. if the struct does not have a property with name *PropertyName*, create a property with name *PropertyName* and value undefined.
 - 2.2. assign the value of *Literal* to the property with name *PropertyName*
3. Return a reference to the struct built in 2.

Informative

The StructLiteral syntax is a subset of the ObjectLiteral syntax in ECMA-262 section 11.1.5, with a somewhat different semantics. In particular:

- A StructLiteral contains only Literals, no expressions
- The PropertyName can only be an identifier, not a string or numeric literal
- The resulting value is not an object but a reference to a struct

Examples

Below are some examples using struct literals. Note that these are primarily given for illustration purposes of struct literals, not as examples of good style of using structs in semantic interpretation tags.

The following line creates a struct with two properties

```
drink = {kind:"tea", portion:"large"};
```

Struct Literals can contain other Struct literals, and Struct properties can have different types:

```
rectangle = { points:[{x:15, y:23}, {x:30, y:70}], line:{size:3, color:"green"} };
```

5.1.5. The Grouping Operator

The production *PrimaryExpression* : (*SemanticExpression*) is evaluated as follows:

1. Evaluate *SemanticExpression*. This may be of type reference.
2. Return Result(1).

Informative

The grouping operator is defined as in ES 11.1.6 but works on SemanticExpressions instead of on Expressions.

5.2. Call Expressions**Normative**

```
CallExpression:
  CallIdentifier ( SemanticExpression )
  ListCallIdentifier ( SemanticExpression , SemanticExpression )
  StringCallIdentifier ( SemanticExpression , SemanticExpression , SemanticExpression )

CallIdentifier:: one of
  String Boolean Number Integer

ListCallIdentifier:: one of
  Append Concat

StringCallIdentifier:: one of
  Substring
```

CallExpression syntax is a subset of ECMA-262 *CallExpression* syntax in ES 11.2:

- *CallExpression* doesn't contain *MemberExpression* but *CallIdentifier*
- *CallExpression* doesn't contain *CallExpression*
- No property accessors
- *ArgumentList* has *SemanticExpression*, not *AssignmentExpression*

By consequence, *CallExpression* syntax is also a subset of ECMA-262 *LeftHandSideExpression* syntax in ES 11.2

5.2.2. Conversion to String

The production *CallExpression* : **String** (*SemanticExpression*) is evaluated as follows:

1. Evaluate *SemanticExpression*
2. Call the ToString operator (see 4.9.5) with as argument Result(1)
3. Return Result(2)

5.2.3. Conversion to Boolean

The production *CallExpression* : **Boolean** (*SemanticExpression*) is evaluated as follows:

1. Evaluate *SemanticExpression*
2. Call the ToBoolean operator (see 4.9.2) with as argument Result(1)
3. Return Result(2)

5.2.4. Conversion to Number

The production *CallExpression* : **Number** (*SemanticExpression*) is evaluated as follows:

1. Evaluate *SemanticExpression*
2. Call the ToNumber operator (see 4.9.3) with as argument Result(1)
3. Return Result(2)

5.2.5. Conversion to Integer

The production *CallExpression* : **Integer** (*SemanticExpression*) is evaluated as follows:

1. Call Number with as argument *SemanticExpression*
2. Call the ToInteger operator (see 4.9.4) with as argument Result(1)
3. Return Result(2)

5.2.6. Appending to List and Concatenating Lists

Informative

The Append operator creates a new list by copying the list given as the first argument and appending one element to it. The copying of the list is a shallow copy, that is: for list elements holding a value of primitive types (Undefined, Boolean, Null, String, Number) the value is copied to the new list, for elements holding a value of type Struct or List the reference to the struct or list is copied to the new list. The newly created list thus holds a reference to the same Struct or List in the original list, and the copying is not recursively applied to such Struct or List.

The Concat operator similarly creates a new list and fills it by shallow copying the two lists that it takes as arguments.

Note that this may create circular references, e.g. in the following sequence of statements:

```
{ x={c:l}; a = Append([],x); x.b =Append(a,4) }
```

Now x.b contains a list in which the first element holds a reference to x itself. Since a list and struct are never copied by deep copy, such circular references are expected to do no harm, except when the information needs to be serialized for transmission outside of the semantic tag processor.

The first argument to Append and Concat is converted to a list if it is not of type list already. When the first argument is undefined or null it is converted to the empty list.

```
$x = $y { p= Append([], $y ) };
$x = $y { p= Append(undefined, $y ) };
```

Appending the undefined value to a list does insert undefined at the end of the list.

The second argument to Concat is converted to a list if it is not of type list already. When the second argument is undefined or null it is converted to the empty list, and thus doesn't add any element.

Normative

The production *CallExpression* : **Append** (*SemanticExpression*, *SemanticExpression*) is evaluated as follows:

1. Evaluate the first *SemanticExpression*
2. Evaluate the second *SemanticExpression*
3. If Result(1) is not of type List, goto step 7
4. Create an internal list whose length is one greater than the length of the list in Result(1)
5. Copy all elements of Result(1) onto this list, in order. Copying of primitive types means copying the value of the element, copying of complex types means copying the reference to the complex value.
6. Goto step 10
7. If Result(1) is null or undefined, create an internal list with one element and goto step 10
8. If Result(1) is of type Number, Boolean or String, create an internal list with two elements
9. Copy GetValue(Result(1)) into the first element of this list.
10. Copy Result(2) into the last element in the internal list created in step 5, 7, or 9
11. Return result(10)

The production *CallExpression* : **Concat** (*SemanticExpression*, *SemanticExpression*) is evaluated as follows:

1. Evaluate the first *SemanticExpression*
2. Evaluate the second *SemanticExpression*
3. Let n be 0
4. If Result(2) is of type List, let n be the length of the list in Result(2)
5. If Result(2) is of type Boolean, Number, or String, let n be 1
6. If Result(1) is not of type List, goto step 11
7. Let p be the length of the list in Result(1)
8. Create an internal list whose length is $p + n$
9. Copy all elements of Result(1) onto this list, in order. Copying of primitive types means copying the value of the element, copying of complex types means copying the reference to the complex value.
10. Goto step 17
11. If Result(1) is not null or undefined, goto step 15
12. Let p be 0
13. Create an internal list with length n
14. Goto step 17
15. Let p be 1
15. Create an internal list with $n + 1$ elements
16. Copy GetValue(Result(1)) into the first element of this list.
17. Use the list created in steps 9, 13, or 16
18. If Result(2) is not of type List, goto step 21
19. Copy all elements of Result(2) onto Result(17), in order, starting at element $p + 1$. Copying of primitive types means copying the value of the element, copying of complex types means copying the reference to the complex value.
20. Return Result(19)
21. If Result(2) is not of type Boolean, Number or String, return Result(17)
22. Copy GetValue(Result(2)) into element $p + 1$ of Result(17)
23. Return Result(22)

5.2.7. Substring selection

Informative

The *CallExpression* Substring takes three arguments, string, start and end, and returns a substring of the result of converting the string argument to type String, starting from character position start and running to, but not including, character position end of the string (or through the end of the string if end is undefined). The result is a string value.

If either of the start or end argument is **NaN** or negative, it is replaced with zero; if either argument is larger than the length of the string, it is replaced with the length of the string.

If *start* is larger than *end*, they are swapped.

The behavior of substring is similar to the behavior defined for the substring prototype of the string object in ECMA-262 section 15.5.4.15, but here it operates on string type, not string object, and it is a *CallExpression*, not a prototype.

Normative

The production *CallExpression* : **Substring** (*SemanticExpression*, *SemanticExpression*, *SemanticExpression*) is evaluated as follows:

1. Call ToString, giving the first *SemanticExpression* as argument
2. Compute the number of characters in Result(1)
3. Call ToInteger with the second *SemanticExpression* as argument
4. If the third *SemanticExpression* evaluates to undefined, use Result(2), else use ToInteger with as argument the third *SemanticExpression*
5. Compute min(max(Result(3), 0), Result(2)).
6. Compute min(max(Result(4), 0), Result(2)).
7. Compute min(Result(5), Result(6)).
8. Compute max(Result(5), Result(6)).
9. Return a string whose length is the difference between Result(8) and Result(7), containing characters from Result(1), namely the characters with indices Result(7) through Result(8)-1, in ascending order.

5.3 Unary Operators

Normative

```
UnaryExpression:
  PrimaryExpression
  CallExpression
  + UnaryExpression
  - UnaryExpression
  ! UnaryExpression
```

UnaryExpression syntax is a subset of the ECMA-262 UnaryExpression syntax (ES 11.4):

- no PostfixExpression but CallExpression and PrimaryExpression (CallExpression and PrimaryExpression syntax are both subset of PostfixExpression syntax)
- no **delete**, **void**, **typeof**
- no prefix increment or decrement
- no bitwise NOT operator

The production *UnaryExpression* : + *UnaryExpression* is evaluated as described in ES 11.4.6

The production *UnaryExpression* : - *UnaryExpression* is evaluated as described in ES 11.4.7

The production *UnaryExpression* : ! *UnaryExpression* is evaluated as described in ES 11.4.8

In all these evaluations, the evaluation of ToNumber is done according to the definitions in section 4.9 (and not directly as defined in ES 9.3)

5.4. Multiplicative Operators

Normative

```
MultiplicativeExpression:
  is defined by reference to MultiplicativeExpression in ECMA-262 section 11.5,
  which is copied here for informative purposes only:
  MultiplicativeExpression:
    UnaryExpression
    MultiplicativeExpression * UnaryExpression
    MultiplicativeExpression / UnaryExpression
    MultiplicativeExpression % UnaryExpression
```

The productions of MultiplicativeExpression are evaluated as described in ES 11.5, 11.5.1, 11.5.2, and 11.5.3.

In all these evaluations, the evaluation of ToNumber is done according to the definitions in section 4.9 (and not directly as defined in ES 9.3)

5.5. Additive Operators

Normative

AdditiveExpression:
is defined by reference to AdditiveExpression in ECMA-262 section 11.6, which is copied here for informative purposes only:
AdditiveExpression:
 MultiplicativeExpression
 AdditiveExpression + *MultiplicativeExpression*
 AdditiveExpression - *MultiplicativeExpression*

The productions of *AdditiveExpression* are evaluated as described in ES 11.6, 11.6.1, 11.6.2, and 11.6.3.

In all these evaluations, the evaluation of *ToPrimitive*, *ToNumber* and *ToString* is done according to the definitions in section 4.9 (and not directly as defined in ES 9, 9.1, 9.3 and 9.8)

5.6. Relational Operators

Normative

RelationalExpression:
 AdditiveExpression
 RelationalExpression < *AdditiveExpression*
 RelationalExpression > *AdditiveExpression*
 RelationalExpression <= *AdditiveExpression*
 RelationalExpression >= *AdditiveExpression*

RelationalExpression syntax is a subset of the ECMA-262 *RelationalExpression* syntax described in ECMA-262 section 11.8 on Relational Operators (note that the ECMA-262 specification has a typo error on the numbering of the section Relational Operators, it numbers it 11.4). In particular:

- no *ShiftExpression* but *AdditiveExpression* (*AdditiveExpression* syntax is a subset of ES *ShiftExpression* syntax)
- no **instanceof**
- no **in**

The *RelationalExpression* productions are evaluated as described in ECMA-262 sections 11.8, 11.8.1, 11.8.2, 11.8.3, 11.8.4 and 11.8.5, except that the contained *AdditiveExpression* is evaluated instead of the contained *ShiftExpression*.

In all these evaluations, the evaluation of *ToPrimitive*, *ToNumber* and *ToString* is done according to the definitions in section 4.9 (and not directly as defined in ES 9, 9.1, 9.3 and 9.8)

5.7. Equality Operators

Normative

EqualityExpression:
is defined by reference to EqualityExpression in ECMA-262 section 11.9, which is copied here for informative purposes only:
EqualityExpression:
 RelationalExpression
 EqualityExpression == *RelationalExpression*
 EqualityExpression != *RelationalExpression*

EqualityExpression syntax is a subset of the ECMA-262 *EqualityExpression* syntax described in ES 11.9. In particular:

- no strict equals and strict does-not-equal operators

The *EqualityExpression* productions are evaluated as described in ECMA-262 sections 11.9, 11.9.1, 11.9.2, and 11.9.3

In all these evaluations, the evaluation of `ToPrimitive` and `ToNumber` is done according to the definitions in section 4.9 (and not directly as defined in ES 9, 9.1, and 9.3)

Note: The equality on `Struct` and `List` is not testing that the elements are the same but only that the two expressions evaluate to the same reference. For example:

```
a=[1,2]; b=[1,2]; c=a;
```

a==b will return false, because although the elements are the same, they are referencing separate lists

a==c will return true, because both a and c hold a reference to the same list.

This is the same behavior as in ECMAScript comparison of objects, and Array Objects in particular.

5.8. Binary Logical Operators

Normative

```
LogicalANDExpression:
  EqualityExpression
  LogicalANDExpression && EqualityExpression

LogicalORExpression:
  is defined by reference to LogicalORExpression in ECMA-262 section 11.11,
which is copied here for informative purposes only:
  LogicalANDExpression
  LogicalORExpression || LogicalANDExpression
```

`LogicalANDExpression` syntax is a subset of the ECMA-262 `LogicalANDExpression` syntax described in ECMA-262 section 11.11. In particular:

- no `BitwiseORExpression` but `EqualityExpression` (`EqualityExpression` syntax is a subset of ECMA-262 `BitwiseORExpression` syntax)

The `LogicalANDExpression` and `LogicalORExpression` productions are evaluated as described in ECMA-262 section 11.11, except that the contained `EqualityExpression` is evaluated instead of the contained `BitwiseORExpression`.

In all these evaluations, the evaluation of `ToBoolean` is done according to the definitions in section 4.9 (and not directly as defined in ES 9 and 9.2)

5.9. Conditional Operator

Normative

```
ConditionalExpression:
  LogicalORExpression
  LogicalORExpression ? SemanticExpression : SemanticExpression
```

`ConditionalExpression` syntax is a subset of the ECMA-262 `ConditionalExpression` syntax described in ECMA-262 section 11.12. In particular:

- no `AssignmentExpression` but `SemanticExpression` (`SemanticExpression` syntax is a subset of ECMA-262 `AssignmentExpression` syntax)

The `ConditionalExpression` production is evaluated as described in ECMA-262 section 11.12, except that the contained `SemanticExpression` is evaluated instead of the contained `AssignmentExpression`.

In all these evaluations, the evaluation of `ToBoolean` is done according to the definitions in section 4.9 (and not directly as defined in ES 9 and 9.2)

5.10. Semantic Expression

Normative

```
SemanticExpression:
  ConditionalExpression
```

SemanticExpression syntax is a subset of the ECMA-262 *AssignmentExpression* syntax described in ECMA-262 section 11.13. In particular:

- no production with *AssignmentOperator*

By consequence, *SemanticExpression* syntax is also a subset of ECMA-262 *Expression* syntax in ES 11.14 that lacks the comma operator.

6. Default Assignment

If there are no Semantic Interpretation Tags attached to any of the expansions in the grammar rule expansion that were used to match the utterance, then the value for the text variable of the GRN is also automatically copied into the GRN's attribute (which then becomes of type string).

Examples:

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

no tags: \$drink is either "coke", "pepsi" or "coca cola", and so is \$drink\$.text

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

shorthand tag on one alternative:
now both coke and coca cola result in "coke" on \$drink
\$drink\$.text is still returning coke, pepsi and coca cola

```
$drink = I want to have a {action = "order"}
(coke | pepsi | coca cola {"coke"});
```

possibly flawed rule: returns either action="order", or "coke" if the utterance contained coca cola. If there is at least one tag in the expansions used, then the default mechanism doesn't work. This means coke or pepsi are never automatically assigned to \$drink in this rule.

7. Visibility Rules and order of tag evaluation for ABNF/XML Speech Recognition Grammar Format

This section defines the visibility rules and order of tag evaluation for SI Tags used in the Speech Recognition Grammar Format (ABNF and XML Form). When SI Tags are embedded in other markup languages (e.g. in ngrams), the visibility rules and order of evaluation may be defined differently.

7.1. Logical Parse Structure

The visibility rules and the order of evaluation of semantic interpretation tags are defined in terms of the logical parse structure as defined in Appendix H. [Logical Parse Structure](#) of the Speech Recognition Grammar Specification.

Note that while this appendix is informative for the Speech Recognition Grammar Specification, it is normative for the Semantic Interpretation specification. This does not imply that grammar processors must implement a logical parse structure, nor that ambiguities or recursion should be handled in any specific way over what is required for a conformant speech recognition grammar processor. The Logical parse structure is only a means to illustrate the order of evaluation and visibility rules for Semantic Interpretation

Tags. Implementations are not required to expose the logical structure and may use different internal representation as long as these yield the results described here.

The Logical Parse Structure is a formal syntax for describing the sequence and relation of tags and rule references to the tokens that are input to the grammar processor.

The Logical Parse output is represented as an array of output entities **en**, e.g. **[e1, e2, e3]**.

Output entities can be one out of three kinds:

- a token, represented as a string holding the literal matching the input to the processor
- a tag, represented as a semantic interpretation tag in curly braces
- a rule reference, represented using [the ABNF form for rule references](#), followed by an array with the output entities generated from that rule reference

Appendix H of the Speech Recognition Grammar Specification contains a full description of how to create the logical parse on a grammar for a given input to a grammar processor.

Example

The sentence "turn the heating off" on the following grammar

```
root $command;
$command = (set | turn) $object $state {$.o=$object;$.s=$state};
$state = (to|$NULL) (on {1} | off {0} | warm {"w"} | cool {"c"} | cold {"c"});
$object = [the] (heating | cooling) {"airco"} | radio {"radio"} | lights {"lights"};
```

would result in the logical parse

```
[ $command [turn,
             $object [the,
                      heating,
                      {"airco"}],
             $state [off,
                     {0}],
                     {$.o=$object, $.s=$state}]
]
```

7.2. Flat Parse List

The logical parse structure is a tree-like structure that shows all terminals, tags and rule references governed by a given rule. This tree can also be represented in a flattened list of parses for every grammar rule application.

The flat parse list for a given rule application is represented as
the rule name followed by a sequence number in parenthesis and a colon
a list of output elements

The output elements are as in the logical parse structure, except that rule references are represented without an output array but followed by a sequence number in parenthesis.

Examples

The equivalent flat parse lists for the above example are:

```
$command(1): turn, $object(1), $state(1), {$.o=$object;$.s=$state}
$object(1): the, heating, {"airco"}
$state(1): off, {0}
```

Following example illustrates the use of the sequence number for rules that are applied more than once:

```

root $a;
$a = ($b)<1-> $c (t1)<0-1> {tag1} | $d {tag2};
$b = t2 | t3 {tag3} | t4;
$c = (t5 {tag5})<1-2>;
$d = t6 $c;

```

Given the input "t2 t3 t5 t5", the logical parse structure is:

```

[$a[ $b[t2], $b[t3, {tag3}], $c[t5, {tag5}, t5, {tag5}], {tag1}]

```

Flat parse lists per rule application:

```

$a: $b(1), $b(2), $c(1), {tag1}
$b(1): t2
$b(2): t3, {tag3}
$c(1): t5, {tag5}, t5, {tag5}

```

7.3. Visibility Rules, Scope

Informative

This description is informative. The normative behavior is contained in the definition of `ReadOnlyPropertyRef` in section 3.4.2.

The attribute and properties of the GRN are shared in all tags in the grammar rule expansions for that grammar rule definition and reference the same attribute or properties:

```

$rule = $a {x=1} ($b {x=3} | ($c {x=x+1})<1->);
// all x's refer to same $.x

```

Ambiguous reference to a GRR attribute or property values or text variables resolves to the closest matching GRR before the tag. The closest matching GRR before the tag is defined as the GRR with the corresponding name that appears before the tag in the flat parse list as defined in section 7.2, if any.

Reference to a GRR value returns undefined if there is no matching GRR before the tag.

Examples

The following rule contains two GRRs to the same rule (\$city).

```

$fromto = from $city {fromcity=$city.name} to $city {tocity=$city.text};

```

To determine which of the GRR instances the tags refer to, we can build the flat parse for \$fromto, which is always of the form:

```

$fromto: from, $city(1), {fromcity=$city.name}, to, $city(2), {tocity=$city.text}

```

From this it follows that \$city.name in the first tag refers to the first GRR \$city in the rule, and that the reference to \$city.text in the second tag is to the second GRR named \$city.

In the following rule, the flat parse is depending on whether the input matches the optional GRR \$b:

```

$a = $b [$b] {x=$b.x};

```

The two possible flat parses are:

```

$a: $b(1), {x=$b.x}
$a: $b(1), $b(2), {x=$b.x}

```

The reference `$b.x` in the tag will thus refer to either the first or the last `$b` in the rule, depending on whether the optional rule `$b` was matched in the input.

The rule below contains a tag that contains a couple of references to GRRs that are undefined since there is no GRR with that name before the tag in the flat parse:

```
$a = $b [$c] {x=$c; y=$d; z=$e} $e;
```

The two possible flat parses are:

```
$a: $b(1), {x=$c; y=$d; z=$e}, $e(1)
$a: $b(1), $c(1), {x=$c; y=$d; z=$e}, $e(1)
```

This means that:

- `$.x` is undefined if `$c` didn't match in the utterance
- `$.y` is undefined because `$d` is not in the rule expansion at all
- `$.z` is undefined because `$e` doesn't appear before the tag

7.4. Order of tag execution

Within a single SI Tag, the order of evaluation is determined by the definitions for *AssignmentList* and *SimpleAssignment* in section 3.3.

The order of evaluating multiple SI Tags within a grammar rule is the order in which the SI Tags appear in the flat parse list for that rule application. The flat parse list also determines how many SI elements will be generated from an SI tag that occurs in a grammar rule. Every SI Tag element in a flat parse list is evaluated exactly once.

The order in which grammar rule applications are taken to process the SI Tags in their corresponding flat parse, is not fixed, but must be such that an SI Tag in a flat parse is evaluated after the flat parses for all rule references in this flat parse before this SI Tag have been evaluated.

Example

Given the following rules

```
$a = $b {y=$b.x} [$b {y=y+$b.x}];
$b = foo {x=1} (bar {x=3} | (boo {x=x+1})<1->);
```

The value of `$.y` in rule `$a` for a few input sentences with these rules:

```
input: foo boo boo boo
flat parses:
$a: $b(1), {y=$b.x}
$b(1): foo, {x=1}, boo, {x=x+1}, boo, {x=x+1}, boo, {x=x+1}
Result: $.y = 4
```

```
input: foo bar foo boo
flat parses:
$a: $b(1), {y=$b.x}, $b(2), {y=y+$b.x}
$b(1): foo, {x=1}, bar, {x=3}
$b(2): foo, {x=1}, boo, {x=x+1}
Result: $.y = 5
```

7.5. Examples

```
1/
$a = $b $c { x = $b.x + $c.x; };
$b.x and $c.x refer to the resp. GRR properties
2/
```

```

$a = $b { x = $b.x + $c.x; } $c;
$c.x causes run-time error because used to the left of $c

3/
$a = [ $b ] $c { x = $b.x + $c.x; };
$b.x evaluates to the x property of $b if $b matched on the input utterance.
Otherwise it causes run-time error
A safer way to write this rule could be (assuming x is of type number):
$a = {x=0} [$b {x=$b.x}] $c {x=x+$c.x};

4/
$a = $b<1-> $c { x = $b.x + $c.x; };
$b.x evaluates to the last occurrence of $b in the count.
If the purpose was to add or concatenate over each occurrence of $b, it should be written as:
$a = ($b {x = x + $b.x})<1-> $c { x = x + $c.x; };

4a/
$a = $b<1-> $c { x = $b + $c.x; };
Similar as 4/, should be e.g.
$a = ($b {x = x + $b})<1-> $c { x = x + $c.x; };

4b/
$a = ($b $d)<0-> $c { x = $b + $c.x; };
Similar: $b evaluates to the last occurrence of $b in the expansion ($b $d)<0->, if any.
Otherwise undefined.

5/
$a = ( $b | $c ) { x = $b.x + $c.x; };
Either $b.x or $c.x will cause a run-time error depending on the input utterance.
This is better written as:
$a = ( $b {x=$b.x} | $c {x=$c.x} );

5a/
$a = ( $b a | a $b ) $c { x = $b.x + $c.x; };
$b.x refers to whichever $b actually matched.

6/
$a = ($b | $c) ($d | $e) { x=($b.x + $c.x) * ($d.x + $e.x); };
One of the operands to every addition causes a run-time error
here depending on the input utterance.
This rule should better be rewritten, e.g. as
$a = ($b {x=$b.x} | $c {x=$c.x}) ($d {x=x*$d.x} | $e {x=x*$e.x}) ;

7/
$a = $b | $c { x = $b.x + $c.x; };
Evaluation of $b.x always causes a run-time error
because the expression will be evaluated only when $c matches, not $b
(When $b matches the default assignment would cause $=$b$.text).
A more useful rule could be:
$a = $b { x = $b.x } | $c { x = $c.x };

8/
$a = $b [ $c { x = $b.x + $c.x; } ];
The expression is only evaluated if $c matches; in that case both $b and $c are defined.

9/
$a = $b ( $c { x = $b.x + $c.x; } )<1-> ;
The expression is evaluated for every occurrence of $c.
Note that this will actually result in $b.x to be added to $c.x for the last occurrence of $c
because every evaluation will overwrite the previous result.

10/
$a = $b ( $c { x = $b.x + $c.x; } )<0->;
Same effect as 9/, except that now the expression is not evaluated if $c didn't match once.

11/
$digits = {ds="" } ( $digit { ds = ds + $digit; } )<1-> ;
$digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
These rules do the obvious concatenation of digits.
Note that ds is first initialized to "" because otherwise in the first evaluation
of the expression, ds would be undefined and cause a run-time error

12/
$a = $b $c { $$ };

```

```

$$ resolves to $c.

13/
$a = $b c { $$ };
$$ resolves to $b.

14/
$a = b c { $$ };
$$ can't be resolved and causes run-time error.

15/
$x = $a ($b | $c) {$$};
If $b matches, $$ resolves to $b. If $c matches, $$ resolves to $c.
This is equivalent to $x = $a ($b {$$} | $c {$$}).

16/
$x = $a [$b] {$$};
$$ resolves to $b, if $b matches, if not, it resolves to $a.
The effect is equivalent to $x = $a {$$} [$b {$$}].

17/
$x = $a<1-> {$$};
$$ resolves to the last occurrence of $a.
The effect is equivalent to $x = ($a {$$})<1->.

```

8. Example Grammar with Semantic Interpretation Tags

Example in ABNF Form:

```

$order = I would like a $drink {drink.liquid = $drink.type;
        drink.drinksizesize = $drink.drinksizesize}
        and $pizza {pizza=$pizza};
// two properties on $order, both are structs
// drink was passed property by property to change a property name
// pizza is passed as whole struct

$kindofdrink = coke | pepsi | coca cola {"coke"};

$foodsize = {"medium"}
[small | medium | large | regular {"medium"}];
// medium is default if nothing said

$stops = $stop {Append([], $stop)}
        (and $stop {Append($, $stop)})<1-> ;
// construct list of toppings, return list

$stop = anchovies | pepperoni | mushroom {"mushrooms"} | mushrooms;

$drink = $foodsize $kindofdrink {drinksizesize=$foodsize; type=$kindofdrink };
// two named properties (drinksizesize and type) on left hand side attribute

$pizza = $number $foodsize {pizzasizesize=$foodsize; number=$number} pizzas
        with $stops {topping=$stops};
// three properties on $pizza's attribute

$number = (a | one){1} | two {2} | three {3};

```

On the above grammar, the following utterance

"I would like a coca cola and three large pizzas with pepperoni and mushrooms."

Would create following struct attribute on \$order:

```

{
  drink: {
    liquid:"coke"
    drinksizesize:"medium"
  }
  pizza: {
    number: "3"
  }
}

```

```

    pizzasize: "large"
    topping: [ "pepperoni" "mushrooms" ]
  }
}

```

Example in XML Form:

```

<rule id="order">
  I would like a
  <ruleref uri="#drink"/>
  <tag> drink.liquid=$drink.type; drink.drinksiz=$drink.drinksiz</tag>
  and
  <ruleref uri="#pizza"/>
  <tag> pizza=$pizza </tag>
</rule>

<rule id="kindofdrink">
  <one-of>
    <item> coke </item>
    <item> pepsi </item>
    <item> coca cola <tag> 'coke' </tag> </item>
    <!-- quote for string constant -->
    <!-- note tag sits inside item -->
  </one-of>
</rule>

<rule id="foodsize">
  <tag> 'medium' </tag> <!-- no more need for NULL -->
  <item repeat="0-1">
    <one-of>
      <item> small </item>
      <item> medium </item>
      <item> large </item>
      <item> <tag> 'medium' </tag> regular </item>
    </one-of>
  </item>
</rule>

<rule id="tops">
  <ruleref id="#top"/>
  <tag> Append([], $top) </tag>
  <item repeat="1-1">
    and
    <ruleref id="#top"/>
    <tag> Append($, $top) </tag>
  </item>
</rule>

<rule id="top">
  <one-of>
    <item> anchovies </item>
    <item> pepperoni </item>
    <item> <tag>'mushrooms'</tag> mushroom </item>
    <item> mushrooms </item>
  </one-of>
</rule>

<rule id="drink">
  <ruleref id="#foodsize"/>
  <ruleref id="#kindofdrink"/>
  <tag> drinksiz=$foodsiz; type=$kindofdrink </tag>
</rule>

<rule id="pizza">
  <ruleref id="#number"/>
  <ruleref id="#foodsiz"/>
  <tag> pizzasiz=$foodsiz; number=$number </tag>
  pizzas with
  <ruleref id="#tops"/>
  <tag> topping=$tops </tag>
</rule>

```

```

<rule id="number">
  <one-of>
    <item>
      <tag> 1 </tag>
      <one-of>
        <item> a </item>
        <item> one </item>
      </one-of>
    </item>
    <item> <tag>2</tag> two </item>
    <item> <tag>3</tag> three </item>
  </one-of>
</rule>

```

9. Future study

9.1 Other shadow variables

It is possible to extend the proposal to other built-in variables than the text variable, such as the list of tokens, the unmodified string, the confidence values, etc. This is left for a future version, if any.

Acknowledgments

This document was written with the participation of and contributions from

Dominique Boucher, Locus Dialogue

Dan Burnett, Nuance Communications

Andrew Hunt, SpeechWorks International

Stefan Krause, Philips

Bruce Lucas, IBM

Mitsuru Oshima, General Magic

Jan Verhasselt, Lernout & Hauspie Speech Products

References

[ECMA]

ECMA Organization - Standardizing Information and Communication Systems

<http://www.ecma.ch>

[ECMA-262]

ECMAScript Language Specification, 3rd Edition - December 1999, published by ECMA.

<http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM>

[N-grams]

Stochastic Language Models (N-Gram) Specification, W3C Voice Browser Activity

<http://www.w3.org/TR/ngram-spec>

[NLSML]

Natural Language Semantics Markup Language for the Speech Interface Framework, W3C Voice Browser Activity

<http://www.w3.org/TR/nl-spec/>

[SRGF]

Speech Recognition Grammar Specification for the W3C Speech Interface Framework, W3C Voice Browser Activity

<http://www.w3.org/TR/speech-grammar>

[Voice]

W3C Voice Browser Activity

<http://www.w3.org/Voice/>

[VoiceXML]

Voice Extensible Markup Language (VoiceXML) Version 2.0, W3C Voice Browser Activity

<http://www.w3.org/TR/voicexml20>