NUANCE PROFESSIONAL SERVICES I Version 1.1 I April 30, 2008

SREC eScript Specification

DOCUMENT HISTORY

Date	Revised by	Approved by	Version	Summary of Changes
7-Oct-2003	Alex de Marco		0.1	Initial internal version.
29-Mar-2007	Jean Dahan		0.2	Changed to Nuance , replaced NR_ with SR_
18-Apr-2007	Jean Dahan		1.0	Updated to reflect updates to grammar specification
30-Apr-2008	Jean Dahan		1.1	Updated to reflect changes to <tag>

TABLE OF CONTENTS

1 OVERVIEW

The eScript language is intended to be a very simple language for expressing semantic information in W3C XML Grammar Format documents. It is inspired by JavaScript/ECMAScript, Babel, and the W3C Semantic Information for Speech Recognition Working Draft. Semantic information is embedded in XML grammars, and this information is used to determine the *meaning* of recognized utterances. For example, the transcript “tune radio ninety seven point three” may be interpreted as {SYSTEM: radio, COMMAND: tune, PARAM: 97.3}.

The grammars used by the Embedded Speech Recognizer (ESR/SREC) conform to the W3C XML Grammar Format standard, but require the use of eScript to express semantic information. Since ESR uses the standardized grammar format and eScript is very similar to ECMAScript/JavaScript, we are able to leverage the power of SpeechWorks Open Speech Recognizer (OSR) offline tools for grammar development. Moreover, some of the grammars already developed for OSR may be compatible with ESR/SREC, and vice-versa. Here are just a few examples of tool reuse: ‘sgc’ to compile grammars, ‘parseTool’ to generate acceptable phrases (i.e. test cases), and more. We have also developed a very simple tool called ‘parseStringTest’ which takes the test phrases generated by ‘parseTool’ and outputs the semantic interpretation of the phrases. OSR’s ‘parseTool’ and ESR ‘parseStringTest’ together facilitate ESR/SREC grammar development. Out of the scope of this document, but still significant is the fact that ESR generates logs which are compatible with SpeechWorks Open Speech Insight Analysis and Reporting tool, which may also be useful for grammar development and tuning.

The eScript language has only one data type, namely strings. All variables are strings and therefore no type conversion, type casting, or type checking is required. Two basic string operations are allowed – assignment and concatenation. Library functions may be added-on to perform more complex operations on strings, such as date/time operations, mathematical operations, and so on. We provide a limited number of such library functions at this time as a convenience to users. The basic idea is to keep the language very simple, but allow more complex behavior by providing a mechanism for calling external pre-compiled library functions to do the “dirty work”.

2 BASIC PRINCIPLES

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

- semantic information is represented as values associated with non-terminals (non-null values)
- semantic expressions are contained in tags associated with grammar rules, as defined by the W3C XML Grammar Format specification.
- expressions in tags are restricted to assignment statements, with some additional string operators and external function calls.
- the order in which expressions are evaluated depends on the order of words in the recognized utterance.

3 EXPRESSIONS IN TAGS

In this section, the expressions that are allowed in semantic information tags are described. These expressions are a subset of ECMAScript. Therefore, eScripts may be interpreted by an ECMAScript processor assuming that ‘built-in’ functions are defined as ECMAScript functions (somehow… TODO). This is due to the fact that they would no longer be ‘built-in’. The opposite is not true however; the eScript interpreter cannot interpret ECMAScripts.

3.1 Variable Scope

All variables are local to the rule that defines them. Only the variables defined in the root rule eventually become part of the ‘meaning’ returned at the end of processing.

```
<rule id="root">
    <ruleref uri="#TwoStrings" />
     <tag>myResult = TwoStrings.first + TwoStrings.second;</tag>
</rule>

<rule id="TwoStrings">
    <item> hello <tag>first=’bonjour’;</tag> </item>
    <item> world <tag>second=’earth’;</tag> </item>
</rule>

  meaning is
          root.myResult: bonjour earth
```

3.2 Boolean Values Represented as Strings

Boolean values true and false are always represented as strings ‘true’ and ‘false’ respectively.

3.3 String Operators

There are 2 operators allowed for strings. We describe them below.

3.3.1 Assignment =

The assignment operator allows the assignment of a string value to a string variable.

3.3.2 Concatenation +

The concatenation operator allows the one string value to be ‘attached’ or ‘added-on’ to another. The length of the concatenated string returned must not exceed the maximum length for strings. We currently do not support the += operator, but will in the future.

3.4 Conditional Expressions

Conditional Expressions are typically used along with assignment statements. An example is shown below where `condition`. If `condition` evaluates to any string other than ‘undefined’ or ‘false’ (both lowercase), then `myVariable` is assigned the string value of `val1`. Otherwise it is assigned the string value of `val2`.

```
  myRule.MyProperty = (condition ? val1 : val2 )
```

3.5 Built-in Library Functions

Built-in library functions extend the basic functionality of the eScript language. The implementations of these functions are pre-compiled and stored in a library, or in the application itself. These external functions allows for improved performance over script interpretation at runtime (i.e. as in ECMAScript). In the future, new functions may be added to the library if required as will be explained. Only strings may be passed to the functions and the functions always return strings.

3.5.1 String Value Interpretation and Mathematical Operations

Sample applications of these functions are radio tuning and setting the clock time. For example, “tune radio ninety five point nine” requires the addition of ninety (‘90’) and five (‘5’), and “set time quarter to three”, requires subtraction of 1 hour from three o’clock and 15 minutes from 60 minutes.

`add(string,string)`
Returns a string representing the sum of the number values of the two strings passed as parameters.

`subtract(minuend,subtrahend)`
Returns a string representing the difference of minuend minus subtrahend.

4 ESCRIPT PROCESSOR INTERFACE

At the NREC-layer, application developers may interface with the Recognizer to get Semantic Results. Listed below are the functions to do so. Complete documentation on each is provided in the NREC API Specification.

Note that the recognizer has an n-best list of semantic results, and each entry has a number of key-value pairs.

```
ReturnCode SR_RecognizerResultGetKeyCount (SR_RecognizerResult *self, size_t index,
size_t *count)
```
Returns the number of keys in the Semantic Result for the n-best list entry identified by `index`.

```
ReturnCode SR_RecognizerResultGetKey (SR_RecognizerResult *self, size_t nbest, size_t
index, LCHAR *key, size_t *len)
```
Returns the key identifier for the key `index` in the n-best list entry identified by `nbest`.

```
ReturnCode SR_RecognizerResultGetKeyValue (SR_RecognizerResult *self, const size_t
nbest, const size_t index, LCHAR *value, size_t *len)
```
Returns the value associated with key `index` for the n-best list entry indexed by `nbest`.

4.1 Special Keys

This sub-section describes “special” keys that exist in the Recognizer Result. The values of these keys are obtained by calling the functions mentioned previously. The ‘raws’, ‘score’, ‘meaning’, ‘literal’, and ‘conf’ keys are considered "global" keys. Any other key must be prefixed with a scope identifier, that is, the rule name. Hence these other keys are local to the scope of the rules (not global).

`meaning`
Meaning of recognized text. The meaning key is only set for the root rule. It is used by the recognizer in nbest processing, and setting it is one way the recognizer filters what gets on the n-best list. It is important that any phrases that mean the same thing be assigned to the same meaning. If it is not set, then it defaults to the concatenation (in arbitrary order) of all property values in the root rule (Same behavior as OSR.)

`literal`
Raw recognized text. The literal key can be accessed only externally to a rule.

`conf`
Degree of confidence with the accuracy of the top choice in the nbest list (scale of 0 to 1000, 1000 = highest confidence)

`raws`
Weights associated with specific n-best results.

5 FACILITY FOR ADDING EXTERNAL FUNCTIONS

Application developers will be provided with a mechanism for adding functions to eScript. These functions must be written in C and be compiled as part of the application itself rather than the eScript processor. The application programmer must register a dispatcher function with the eScript processor (i.e. function pointer), and subsequently implement this dispatch function to call an appropriate function in the application to carry out a task.

When eScript processing encounters a token in an expression that it does not understand, the application's dispatch function will be called. The dispatch function will call another function to operate on the paramers, and then return the result of the computation (always represented as a string). If the dispatch function fails to dispatch, then an error code is returned to the eScript processor. This is the same mechanism for external functions built into SDX/Babel. We quote from the SDX/Babel documentation below.

ReturnCode Dispatch(LCHAR* name, LCHAR** args, size_t argCount, LCHAR* resultBuffer, size_t* bufLen),

```
where:

name is the name of the function invoked;

args is the list of arguments passed to the function (in the above example, two
strings contained in variables n1 and n2);

argCount is the number of arguments passed to the function (in the above example,
two);

resultBuffer is the buffer in which the result of the operation will be stored;

bufLen is the length of the buffer.

Dispatch returns 0 if the call was completed sucessfully, or the following error
codes:

DISPATCH_ERROR_GENERAL_ERROR = -1
DISPATCH_ERROR_BUFFER_OVERFLOW = 1
DISPATCH_ERROR_BAD_ARGUMENT_COUNT = 2
DISPATCH_ERROR_UNKNOWN_FUNCTION = 3
```

After the Dispatch function is defined, its address needs to be passed to the eScript processor via the **SR_GrammarSetDispatchFunction()**. The Dispatch function must have the following signature, and will in most cases have the following behavior. (For further detail refer to the NREC API Specification.)

ReturnCode Dispatch(LCHAR* name, LCHAR** args, size_t argCount, LCHAR* resultBuffer, size_t* bufLen)

```
{
 if ( strcmp( name, "add" )==0 ) {
       return _Add( args, argCount, resultBuffer, bufLen );
 } else if ( strcmp( name, "sub" )==0 ) {
       return _Sub( args, argCount, resultBuffer, bufLen );
 } ...
 } else {
 return DISPATCH_ERROR_UNKNOWN_FUNCTION;
 }
}

int _Add( char** args, int argCount, char* resultBuffer, int bufLen )
{
 if (argCount != 2) {
 return DISPATCH_ERROR_BAD_ARGUMENT_COUNT;
 } else {
 // convert args[0] and args[1] to integers
 // add the two values
 // convert the result into a string and store in
 // resultBuffer
 // DON'T FORGET TO TERMINATE IT WITH A ZERO!
 // if the result is too long, return
 // DISPATCH_ERROR_BUFFER_OVERFLOW
 }
}

etc.
```

6 EXAMPLES OF ESCRIPT USAGE

In Appendix A, the eScript grammar is provided. Listed below are examples of valid statements. **Note** that white-space characters are ignored during interpretation and that when including several statements within a tag, they must be separated with semi-colons.

```
// The user may or may not actually say ‘FM’ when tuning a radio,
// therefore, if undefined, we set this rule’s property FM to value
// ‘FM’
FM = ( freq_band.FM ? freq_band.FM : ‘FM’ )

// tens is a reference to a rule for words like ‘ten’, ‘twenty’
// ‘thirty’, and so on. digit is a rule reference for ‘one’, ‘two’,
// ‘three’, etc. Rule properties V corrspond to the string values // of
the spoken words. Now, for example if tens.V is ‘90’,
// and digit.V is ‘1’, we add them together to get ‘91’ by calling
// the add() function.
TENS_ONES = add ( tens.V, digit.V )


// Below is a set of statements for constructing a Canadian postal
// code from letters and digits. Notice that PC must be written to
// in the first statement before it is read from. Otherwise, the
// string ‘undefinded’ would be prepended to PC. Also notice that
// a maximum of two operands are allowed in the right hand side of
// the assigmnet. We may support additional operands in the future.
PC = letter.V ;
PC = PC + digit.V ;
PC = PC + letter.V ;
PC = PC + digit.V ;
PC = PC + letter.V ;
PC = PC + digit.V ;
```

Advantages of using such simple statements:

- *Lexical Analysis and Parsing*: No need to build a parse tree for parsing a new expression. When new tokens are required, we simply shift pointers in a string buffer that contains the statements. This reduces execution time and dynamic memory usage normally associated with building a tree with an arbitrary number of arcs/nodes.
- Having a finite number of operands gives us an upper-bound on the number of instructions it will take to process an expression and the amount of memory required for each expression’s interpretation. It is usually preferred to have many small and simple instructions to process rather than few large and complex instructions. This is the concept behind RISC processors.

7 POSSIBLE FUTURE BUILT-IN FUNCTIONS

In the future, we may build-in the following functions if there is a demand for them. We currently do not see a great need for them, but if customers demand such functionality we can, at some point in the future, implement them and include them in the function library.

7.1.1 Substring Extraction

`substr(string, start_index, length)`
Returns a substring of string stating at start_index, up to length.

`substr(string, start_index)`
Returns a substring of string stating at start_index, up to the end of string.

7.1.2 String Value Interpretation and Comparison

`isGreaterThan(string, pivot_string)`
Returns ‘true’ if the number value of string is greater than the pivot_string number value; ‘false’ otherwise.

`isGreaterThanOrEqual(string, pivot_string)`
Returns ‘true’ if the number value of string is greater than or equal to the pivot_string number value; ‘false’ otherwise.

`isLessThanOrEqual(string, pivot_string)`
Returns ‘true’ if the number value of string is less than the pivot_string number value; ‘false’ otherwise.

`isLessThanOrEqual(string, pivot_string)`
Returns ‘true’ if the number value of string is less then or equal to the pivot_string number value; ‘false’ otherwise.

7.1.3 Date / Time Processing

`parseAbsoluteDate(year, month, day)`
Converts year, month, and day strings passed as parameters to a string representing the absolute date.

`parseRelativeDate(year, month, day)`
Converts year, month, and day strings passed as parameters to a string representing the relative date from today. Example: year = ‘+1’, month= ‘0’, day= ‘-1’ returns the date 364 days from now.

`parseAbsoluteTime(hour, minute, second)`
Converts hour, minutes, and seconds strings passed as parameters to a string representing the absolute time.

`parseRelativeTime(hour, minute, second)`
Converts hour, minutes, and seconds strings passed as parameters to a string representing the relative time from now.

`parseOnesTensHundreds(ones, tens, hundreds)`
Converts ones, tens, hundreds strings passed as parameters to a string representing the number in the thousands range. Example: ones = ‘6’, tens= ‘9’, hundreds= ‘19’ returns ‘1996’.

7.1.4 Mathematical Operations

`Multiply(string,string)`
Returns a string representing the product of the number values of the two strings passed as parameters.

`Divide(dividend,divisor)`
Returns a string representing the quotient of the number value of the dividend string divided by the divisor string. The remainder is ignored.

8 HIGH-LEVEL ARCHITECTURAL VIEW

The high-level architecture of the eScript processor is similar to the architecture one would normally expect for a typical compiler or interpreter. For eScript however, key issues are footprint, efficiency, and performance, due to the constraints of embedded systems in general. In addition, it must be single-threaded.

We have four modules and each is actually quite simple. The idea of separating functionality as shown in Figure 1 and explained below will facilitate debugging and future enhancement.

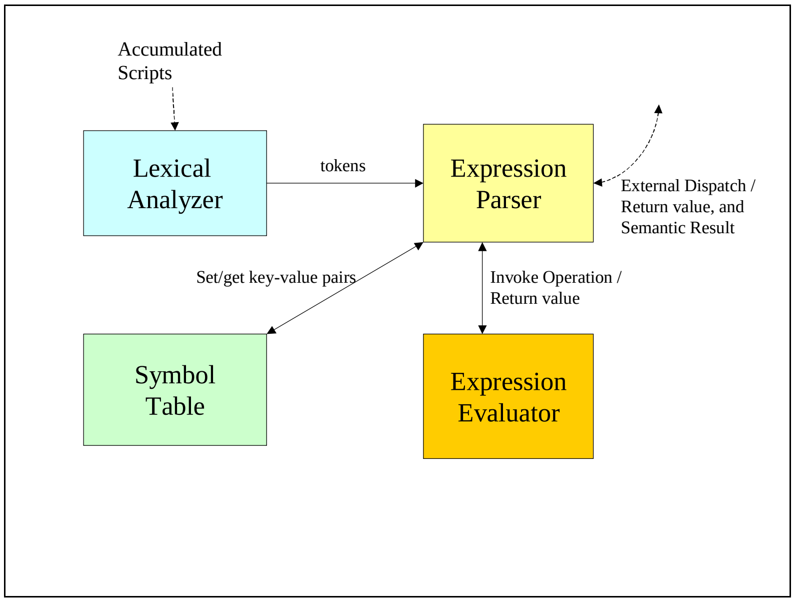

Figure 1: High-level Architecture

Lexical Analyzer: Translates accumulated expressions (i.e. strings) to stream of tokens. Delivers stream of tokens to the Expression Parser. Assumes that rule names have been pre-pended to all properties so that we may have properties with the same name in different rules.

Expression Parser: Determines whether token stream is syntactically correct. Translates tokens to keys and values, which are stored in the Symbol Table. Calls appropriate function(s) in Expression Evaluator or in the application through the dispatch function as explained in section 4. At the end of parsing, returns the value associated with properties in the root rule.

Expression Evaluator: Executes the operations as instructed to by the Expression Parser. Contains a basic function set to carry out operations, namely string assignment, string concatenation, conditional expression evaluation.

Symbol Table: The symbol table is simply a hash table storing key-value pairs.

8.1 Assumptions

Accumulated Script Execution: Scripts, also known as “tag content”, will be accumulated and stored temporarily as grammar rules are fired. Only at the end of grammar parsing shall the eScript processor be invoked to interpret semantic information. This allows memory that was used for parsing the spoken utterance to be reused by the eScript processor after it has been freed by the grammar parser. If we would do both at the same time, grammar processing and semantic processing, this could lead to an overflow of memory bounds, even though the semantic processor’s memory consumption may be small.

Rule Name Pre-pending: An important assumption for the Lexical Analyzer, as stated previously, is that rule names are pre-pended to property names. This task of prepending will be carried out as grammar rules are fired and scripts are accumulated in a temporary buffer. Each assignment in a script will have the rule name pre-pended to property names which are encountered on either side of the assignment operator. This would of course only be done for properties that do not already have rule name references prepended. This is one way for us to ensure that variables (i.e. properties) will remain local to the scope of a rule without the need for dynamically creating new scopes during processing. See Appendix C and below for examples.

```
<rule id=”test”>
   <item> something <tag> PropName=’PropVal’</tag></item>
</rule>
   When this rule is fired, the script that would be accumulated is
   test.PropName = ‘PropVal’
```

APPENDIX A: ESCRIPT GRAMMAR

```
NOTE: Terminal strings are denoted by single quotes.

S → RulePropertyReference AssignmentOperator Expression

RulePropertyReference → RuleNameTerminal . RulePropertyNameTerminal

AssignmentOperator → ‘=’

Expression → StringExpression |
              ConditionalExpression |
               ExternFunctionExpression

StringExpression → Operand |
                    Operand ConcatenationOperator Operand

Operand → RulePropertyReference | ConstantStringTerminal

ConcatenationOperator → ‘+’

ConditionalExpression →
         ‘(‘ RulePropertyReference ‘?’ Operand ‘:’ Operand ‘)’

ExternFunctionExpression → ‘EXTERN_FN_NAME’ ‘(‘ ‘PARAM’ ExtnFnParam ‘)’

ExtnFnParam → ‘,’ ‘PARAM’ | .eps

RuleNameTerminal → ‘RULE_NAME_TERMINAL’

RulePropertyNameTerminal → ‘RULE_PROPERTYNAME’

ConstantStringExpression → ‘CONSTANT_STRING’
```

APPENDIX B: EXAMPLE OF ARBITRARY LENGTH DIGIT STRING

```
<grammar xml:lang="en-us" version="1.0" root="root">

  <rule id="root" scope="public">
     <ruleref uri="#DIGIT" />
           <tag> NUM=DIGIT.V </tag>
     <count number="0+">
           <ruleref uri="#DIGIT"/><tag> NUM=NUM+DIGIT.V</tag>
     </count>
  </rule>

  <rule id="DIGIT">
      <one-of>
      <item> zero <tag> V=’0’</tag></item>
      <item> oh <tag> V=’0’</tag></item>
      <item> one <tag> V=’1’</tag></item>
      <item> two <tag> V=’2’</tag> </item>
      <item> three <tag> V=’3’</tag> </item>
      <item> four <tag> V=’4’</tag></item>
      <item> five <tag> V=’5’</tag></item>
      <item> six <tag> V=’6’</tag></item>
      <item> seven <tag> V=’7’</tag></item>
      <item> eight <tag> V=’8’</tag></item>
      <item> nine <tag> V=’9’</tag></item>
      </one-of>
   </rule>
</grammar>
```

Important Note: Notice that in the root rule, NUM is initialized before it is read from. This ensures that we do not get a NUM that begins with undefined data. For example, if we would not have initialized NUM, and changed the count to 1+, then NUM=NUM+DIGIT.V would have appended digits to an undefined string to give something like ‘undefined1234455677…’. In general, you must initialize strings before appending to them… they are *not* empty by default.

APPENDIX C: EXAMPLE OF FM RADIO TUNER

```
<grammar xml:lang="en-us" version="1.0" root="root">

<!-- This grammar accepts the following sample transcripts:

NOTE: FM radio spectrum is 88-108 MHz. This grammar needs to be fixed so
that it will not accept frequencies outside this range. TODO.

radio set tuner nine nine point five - 99.5
radio tuner eighty nine three FM - 89.3
radio set tuner nine one one - 91.1
radio tune one oh oh dot one - 100.1
radio frequency nine five three FM - 95.3
radio station eight eight dot one - 88.1
radio tune one oh oh nine - 100.9
radio tuner nine five seven FM - 95.7
radio tuner eight eight five FM - 88.5
radio tuner one hundred one five FM - 101.5
radio tune ninety nine nine - 99.9

NOTE: No AM support for now

-->

  <rule id="root" scope="public">
      <ruleref uri="#radio" <tag>"COMMAND=radio.COMMAND"</tag>>
  </rule>


  <rule id="radio">
      <item>radio</item>
      <ruleref uri="#radioSetFreq"
               <tag>"COMMAND = 'RADIO_SET_FREQ ';
                COMMAND = COMMAND + radioSetFreq.FREQ;
               COMMAND = COMMAND + ' ';
               COMMAND = COMMAND + radioSetFreq.BAND;
               "</tag>
  </rule>

  <rule id="radioSetFreq">
      <one-of>
      <item>
          <count number="optional">set</count>
          <one-of>
              <item>frequency</item>
              <item>tuner</item>
              <item>station</item>
          </one-of>
      </item>
```

```
        <item>tune</item>
        </one-of>
        <ruleref uri="#frequency" />
                 <tag> FREQ=frequency.F;
                     BAND=(frequency.B ? frequency.B : 'FM’); </tag>

    </rule>

    <rule id="frequency">
        <ruleref uri="#frequencyFM" />
                 <tag>F=frequencyFM.I + '.' + frequencyFM.D </tag>
        <count number="optional">FM<tag> "B='FM'</tag> </count>
    </rule>

    <rule id="frequencyFM">
      <ruleref uri="#FMIntegerPart"/> <tag> I=FMIntegerPart.V </tag>
      <count number="optional"><ruleref uri="#DOT"/></count>
      <ruleref uri="#oddDIGIT" /> <tag>D=oddDIGIT.V</tag>
    </rule>

    <rule id="FMIntegerPart">
       <one-of>
           <item>
             <ruleref uri="#FMTensOnes”
                <tag>"V=add(FMTensOnes.tens,FMTensOnes.ones)"</tag>
           </item>
           <item><ruleref uri="#FMTens" <tag>"V=FMTens.tens"</tag></item>
       </one-of>
    </rule>

     <rule id="FMTensOnes">
            <item>
            <one-of>
                <item <tag>"tens='80'"</tag>eighty</item>
                <item <tag>"tens='80'"</tag>eight</item>
                <item <tag>"tens='90'"</tag>ninety</item>
                <item <tag>"tens='90'"</tag>nine</item>
                <item <tag>"tens='100'"</tag>one oh</item>
                <item <tag>"tens='100'"</tag>one zero</item>
                <item <tag>"tens='100'"</tag>one hundred</item>
            </one-of>
            <ruleref uri="#noZeroDIGIT" <tag>"ones=noZeroDIGIT.V"</tag>/>
        </item>
     </rule>

     <rule id="FMTens">
            <item>
            <one-of>
                <item <tag>"tens='80'"</tag>eighty</item>
                <item <tag>"tens='90'"</tag>ninety</item>
                <item <tag>"tens='100'"</tag>one hundred</item>
                <item <tag>"tens='100'"</tag>one zero zero</item>
                <item <tag>"tens='100'"</tag>one oh oh</item>
            </one-of>
        </item>
```

```
   </rule>


   <rule id="DOT">
      <one-of>
          <item>dot</item>
      <item>point</item>
      </one-of>
   </rule>

   <rule id="DIGIT">
      <one-of>
      <item><ruleref uri="#evenDIGIT" <tag>"V=evenDIGIT.V"</tag>/></item>
      <item><ruleref uri="#oddDIGIT"  <tag>"V=oddDIGIT.V"</tag>/></item>
      <item><ruleref uri="#zeroDIGIT" <tag>"V=zeroDIGIT.V"</tag>/></item>
      </one-of>
   </rule>

   <rule id="noZeroDIGIT">
      <one-of>
      <item><ruleref uri="#evenDIGIT" <tag>"V=evenDIGIT.V"</tag>/></item>
      <item><ruleref uri="#oddDIGIT"  <tag>"V=oddDIGIT.V"</tag>/></item>
      </one-of>
   </rule>

  <rule id="oddDIGIT">
      <one-of>
      <item <tag>"V='1'"</tag>one</item>
      <item <tag>"V='3'"</tag>three</item>
      <item <tag>"V='5'"</tag>five</item>
      <item <tag>"V='7'"</tag>>seven</item>
      <item <tag>"V='9'"</tag>nine</item>
      </one-of>
   </rule>

   <rule id="evenDIGIT">
      <one-of>
      <item <tag>"V='2'"</tag>two</item>
      <item <tag>"V='4'"</tag>four</item>
      <item <tag>"V='6'"</tag>six</item>
      <item <tag>"V='8'"</tag>eight</item>
      </one-of>
   </rule>

   <rule id="zeroDIGIT">
      <item <tag>"V='0'"</tag>zero</item>
      <item <tag>"V='0'"</tag>oh</item>
   </rule>

</grammar>
```

Important Note: In processing the spoken utterance “radio tune ninety seven point seven”, the grammar processor will accumulate all semantic information (scripts) and eventually pass them to the semantic processor. The accumulation of scripts will be as shown below.

```
tens='90'
V='7'
V=oddDIGIT.V
ones=noZeroDIGIT.V
V=add(FMTensOnes.tens, FMTensOnes.ones)
I=FMIntegerPart.V
V='7'
D=oddDIGIT.V
F=frequencyFM.I + '.' + frequencyFM.D
FREQ=frequency.F;
BAND=(frequency.B ? frequency.B : 'FM');
COMMAND = 'RADIO_SET_FREQ ';
COMMAND = COMMAND + radioSetFreq.FREQ;
COMMAND = COMMAND + ' ';
COMMAND = COMMAND + radioSetFreq.BAND;
COMMAND = radio.COMMAND
```

Notice that the eScript processor would not know how to interpret the third line (among others) because oddDIGIT.V is not explicitly defined anywhere. For this reason, the grammar processor must prepend rule names to properties that do not have rule names associated with them before passing the accumulated scripts to the eScript interpreter. The result is shown below.

```
FMTensOnes.tens='90'
oddDIGIT.V='7'
noZeroDIGIT.V=oddDIGIT.V
FMTensOnes.ones=noZeroDIGIT.V
FMIntegerPart.V=add(FMTensOnes.tens,FMTensOnes.ones)
FrequencyFM.I=FMIntegerPart.V
oddDIGIT.V='7'
frequencyFM.D=oddDIGIT.V
frequency.F=frequencyFM.I + '.' + frequencyFM.D
radioSetFreq.FREQ=frequency.F;
radioSetFreq.BAND=(frequency.B ? frequency.B : 'FM');
radio.COMMAND = 'RADIO_SET_FREQ ';
radio.COMMAND = radio.COMMAND + radioSetFreq.FREQ;
radio.COMMAND = radio.COMMAND + ' ';
radio.COMMAND = radio.COMMAND + radioSetFreq.BAND;
root.COMMAND = radio.COMMAND
```