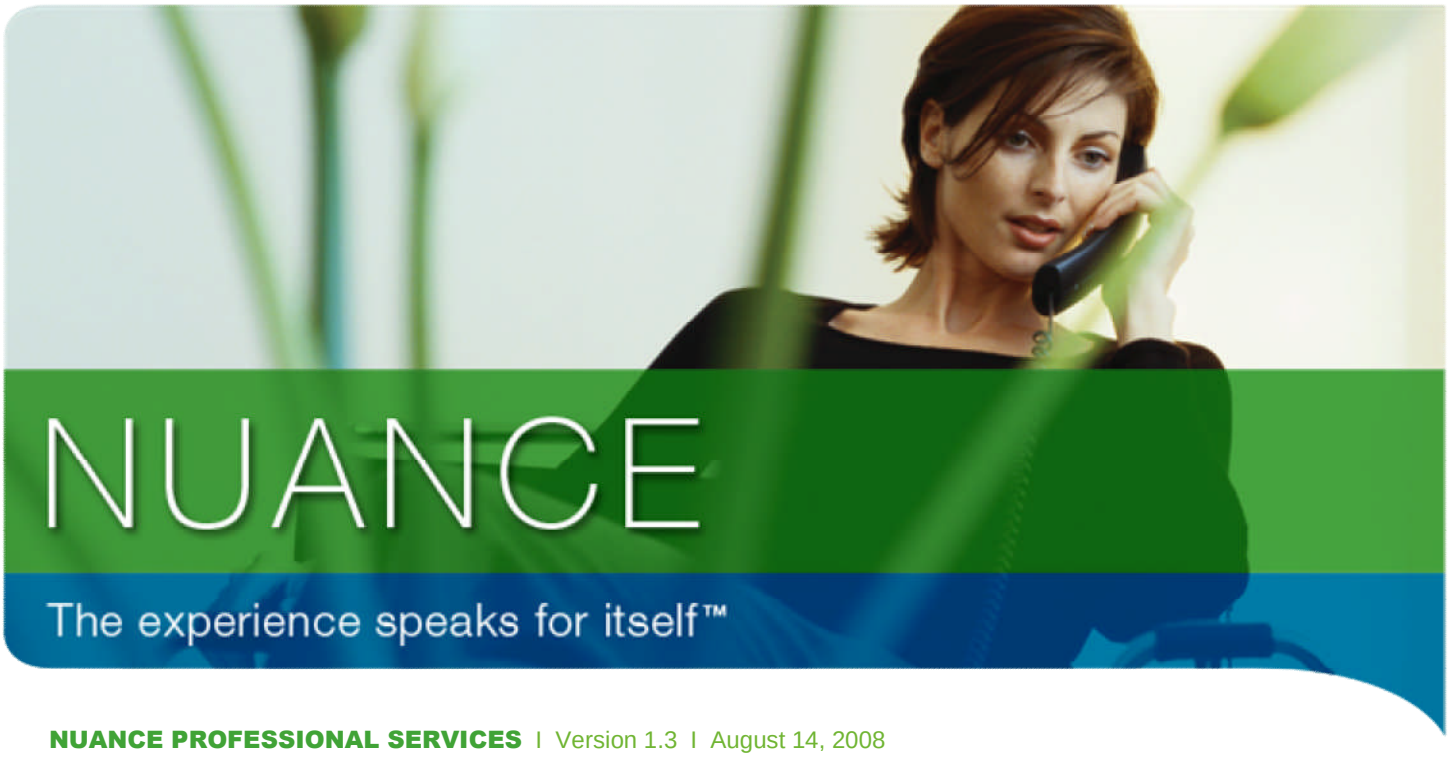



**NUANCE PROFESSIONAL SERVICES** I Version 1.3 I August 14, 2008

# SREC Architecture

August 2008

NUANCE

## DOCUMENT HISTORY

| Date | Revised by | Approved by | Version | Summary of Changes |
|---|---|---|---|---|
| 12/20/2007 | Jean Dahan, Xufang Zhao | | 1.0 | RC-1 version |
| 02/12/2008 | Rabih Majzoub | | 1.1 | Added dynamic word addition documentation |
| 03/07/2008 | Jean Dahan | | 1.2 | Minor updates for dynamic slot allocation |
| 08/14/2008 | Jean Dahan | | 1.3 | Minor updates for enrollment, G2P |

## TABLE OF CONTENTS

1 Introduction ..... 4
2 OVERVIEW OF FEATURES ..... 5
3 SREC BASE Modules ..... 6
3.1 ESR_Portable ..... 6
3.2 ESR_Shared ..... 6
3.3 Audio ..... 6
4 Front End and Acoustic Models ..... 7
4.1 CA_FrontEnd ..... 7
4.2 SR_AcousticModels ..... 7
4.3 Duration model ..... 8
4.4 swiarb file format ..... 9
4.5 swimdl file format ..... 9
4.6 swimdllst file ..... 10
4.7 LDA file format ..... 10
4.8 SR_AcousticState ..... 11
5 SR_Grammar ..... 12
5.1 Text-format grammar files ..... 12
5.2 G2G-format grammar file ..... 13
5.3 Slot-based word and voice-tag addition ..... 13
5.3.1 Dynamic word addition ..... 14
5.3.2 Resetting the slots ..... 15
5.4 SR_SemProc ..... 15
6 Voicetags: SR_Nametag, SR_Nametags ..... 16
6.1 Hmm-Based enrollment ..... 16
6.2 Phonetic-based enrollment ..... 16
7 SR_Vocabulary & G2P engine ..... 17
7.1 Dictionary ..... 17
7.2 G2P Engine ..... 17
7.3 Word-specific phonemes ..... **Error! Bookmark not defined.**
7.4 Phonetic Inventory ..... 17
8 SR_Recognizer ..... 18
8.1 The main search ..... 18
8.2 Support for multiple acoustic models ..... 20
8.3 End of the search and end of utterance timeouts ..... 20
8.4 The nbest search ..... 20
8.5 Confidence value calculation ..... 21
9 GRAMMAR COMPILER ..... 22
10 CONTRIBUTORS ..... 24

# 1 INTRODUCTION

This document describes the internals of the SREC embedded speech recognition engine. This gives a higher level picture of the engine than the code provides, but a much more detailed description than the user guide. This is intended for developers who wish to add capabilities to the engine and/or develop auxiliary tools.

The SREC engine and related tools are licensed under the Apache License, Version 2.0. The code cannot be used except in compliance with the License (http://www.apache.org/licenses/LICENSE-2.0). Developers who have contributed to this engine are listed in the appendix.

# 2 OVERVIEW OF FEATURES

SREC is a phonetic-based speech recognition system intended for use in embedded applications. The API is synchronous.

SREC is a continuous speech phonetic-based speech recognition system. The following features are supported:

- speaker-independent: the SREC acoustic models are trained from a variety of speakers
- phonetic-based: this allows any word to be recognized without previous training; different allophonic decision tree models are trained for each phoneme in context; SREC also uses some whole word models to maximize accuracy for specific words like digits.
- SREC is a continuous speech recognizer; this means that the speaker doesn't have to pause between the words when giving complex commands
- "voicetags"; utterances intended to be used as voicetags are decoded and generate an internal transcription that can be added to a slot
- slot-based word-addition; text words or voicetags (see below) can be added to a slot; this allows functions such as recognition from dynamically changing data such as contact lists
- support for a simple semantic interpretation language (eScript, documented elsewhere) that allows grammar developers to associate grammar specific orthographies and/or synonyms to application actions.
- online Grapheme-to-Phoneme is supported; so text words can be added to a slot without pre-defining the pronunciation;

# 3 SREC BASE MODULES

SREC is organized as a collection of inter-dependent libraries. The public facing API functions begin with SR_ or ESR_. The CA_ and srec_ APIs are internal, and not intended for application development.

## 3.1 ESR_Portable

The portable library abstracts I/O operations, memory management and other OS-dependant functionalities away from users. Currently, the entire ESR product set makes use of this API; including sample code.

Character Represtation: The character representation is defined in this library as LCHAR. LCHAR is currently defined as a narrow “char”, other implementations are possible, but have not been tried.

Memory Allocaction: This library abstracts the memory allocation system, which allow measurements of memory use and leaks.

File System: This library abstracts the file system from the application. This allows SREC and other ESR functions to use pfopen, pfwrite, pfread etc .. type functions even while fopen, fwrite, fread, etc are not supported. This has been tried and tested on the VXWORKS operating system.

ArrayList, PHashTable_t: Some basic forms for these structures is supported here.

## 3.2 ESR_Shared

The ESR_Shared library includes a number of utilities, including higher-level hash tables, arrays, string functions and buffers.

shared/src/CircularBuffer.c
shared/src/CommandLine.c
shared/src/ESR_Locale.c
shared/src/ESR_Session.c
shared/src/HashMap.c
shared/src/HashMapImpl.c
shared/src/Int8ArrayList.c
shared/src/Int8ArrayListImpl.c
shared/src/IntArrayList.c
shared/src/IntArrayListImpl.c
shared/src/lstring.c
shared/src/LStringImpl.c
shared/src/SessionTypeImpl.c

## 3.3 Audio

The audio library consists of reference audio implementation for Linux , Windows and Android. It is used by test programs such as SRecTestAudio to recognize from live audio rather than from files.

# 4 FRONT END AND ACOUSTIC MODELS

## 4.1 CA_FrontEnd

SREC uses standard front-end processing techniques, using fixed-point arithmetic. Frames representing 20ms of speech are generated every 10ms. Pre-emphasis is applied, followed by a Hamming window and FFT. SREC then finds the log energy in each of 26 mel-spaced filters, and runs an inverse cosine transform on these log-energies to get 12 MFCCs. The acoustic models are trained to a specific "highcut" of the mel-spaced filters, which is to say that when the sample rate of the input speech is more than double the highcut then no subsampling is required, the front-end merely ignores the frequency content above the highcut.

Delta and double-delta of MFCCs are then applied, resulting in a 36 dimensional feature vector.

SREC then applies a linear transformation which orders the features according to discriminant power (i.e. LDA, linear discriminant analysis). Of the 36 outputs, SREC retains only 24.

Cepstral-mean normalization is applied on the LDA transformed features. The initial cepstral mean is determined empirically for a test set resembling the data expected from the deployment microphone. The cepstral mean is updated at the end of each recognition, using the speech frames from the previous recognition.

## 4.2 SR_AcousticModels

This class represents a collection of acoustic models. Models may be loaded from or saved to disk or associated with a recognizer. The recognizer generally uses 2 acoustic models, one trained from male speakers and one from females. The two acoustic models have exactly the same allophonic structure, allowing the same allophonic graph to be used for decoding with both.

The cepstral and dynamic coefficients generated by the front-end are passed through a CA_Pattern object, which basically applies a linear discriminant analysis (LDA) matrix transformation to decorrelate the features. The LDA is a diagonalizing transform whose output coefficients are ordered by discriminant power, and we've found that only the first 28 tend to be useful. Thus, the acoustic model means are 28 dimensions wide.
The phoneme table was based on US English phoneme set, and the phoneme set name is "ENU54", version 1.0. There are total 56 phonemes in the ENU54 table; however, in srec acoustic models, the basic phoneme set was reduced to 51 phonemes including "silence", but there are not "environment" phoneme (ENV) and four pause-fill phonemes, Pah, Puh, Pum, and Peh. Furthermore, different from the ENU54 table, almost all phonemes in srec acoustic models have only two states except five diaphones, "OY", "OW", "EY", "AY", and "AW"; these five diaphones have three states in srec acoustic models.

To increase the recognition accuracy for digits and some popular words such as "yes" and "no", "on" and "off", phoneme labels for these words were relabeled using special phoneme symbols in the training corpora and question trees. For example, the word "four" was relabeled as "f0 AH0 r0" instead of "f AWH r"; there are total 29 word-specific phonemes. All word-specific phonemes consist of 2 states except three word-specific diaphones, "AW0", "AY0", and "EY0". The appendix A is the sub-dictionary using relabeled word-specific phonemes.

The final phone table of srec acoustic model includes 51 basic phonemes and 29 word-specific phonemes; the number of total phonemes is 80. The appendix B is the srec phoneme table.

The srec acoustic models are gender-specific model, which means that male models and female models are trained separately using gender-specific data. Both male model and female model are triphone models, and they shared an exactly same decision tree, so the male model and female model have the same number of leaf nodes. The decision tree file has the file extension of .swiarb, and it was created

from a general training corpus that contains both male and female training data. The different states in a phoneme share the completely same tree structure, and leaf nodes that are in the same tree location but different states were concatenated to create a HMM (Hidden Markov Model). The Fig.1 illustrate such tree structure and HMM generation.

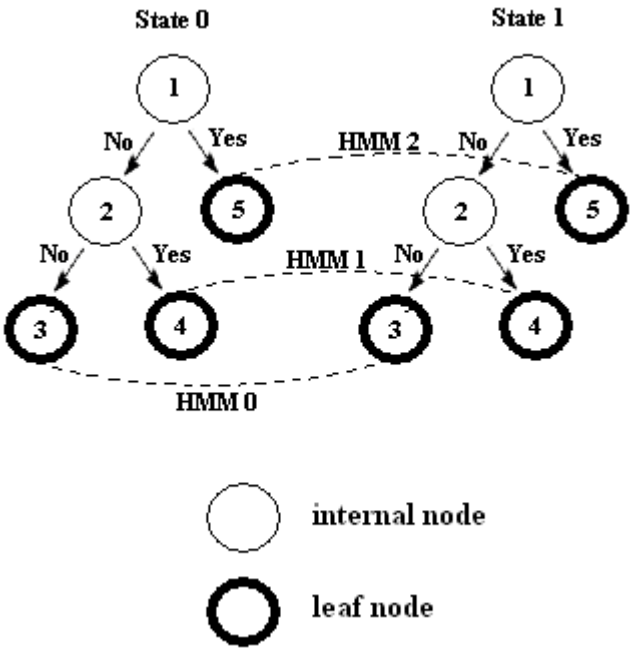



**Fig.1 An example of tree structure and HMM creating**

In srec acoustic models, the Probability Density Function is a Gaussian distribution. Each Gaussian distribution is a component, and some components were mixed together using different mixture weight to create a Pel. The “silence” model is Pel0, and it contains 32 components; other Pels were consisted of 6 components.

The feature dimension in srec acoustic models is 36, which includes 12 MFCCs, 12 delta values, and 12 double delta values.

### 4.3 Duration model

SREC uses an explicit duration model which allow the search to control how long the search state can loop on a given hmm state, aka how long the residency on the state lasts. The duration model is implemented as a cost to be applied if the search is looping on a given state for too long (the longer you stay, the more you must pay), and when the search is trying to leave too early from a given state (the earlier you leave, the more you must pay). The average duration for each state is calculated at acoustic model training time and is stored in the acoustic models. Two hard-coded tables are used to lookup costs associated with looping past the desired average duration and with departing prior to the desired average duration. The tables are determined empirically using a spreadsheet, so as to make the mean of the residency durations approximately equal to the average state duration and the variance approximately equal to half the duration. These tables do not need to be regenerated for new acoustic models, but they are dependent on the CREC.Recogizer.do_skip_even_frames = 1 parameter.

Sources:
srec/clib/swimodel.c

### 4.4 swiarb file format

The swiarb file provides triphone decision trees for each phoneme. Furthermore, the swiarb file provides a duration mean time for each leaf node; the unit of a duration mean is mille-second. If the duration means of a node in decision trees is -1, this node is an internal node and it does not have duration means time.

The swiarb file is a binary file. Tab.2 described the format of a swiarb file, and the column 4 is an example of a typical srec swiarb file; its total size is about 42k bytes.

| Size unit | number | description | Size of a typical srec swiarb file (Byte) |
|---|---|---|---|
| Char * | 1 | Image point | 4 |
| Uint16 | 1 | Image size | 2 |
| Int16 | 1 | The number of phonemes | 2 |
| Struct * | 1 | Pointer of phoneme buffers | 4 |
| Int16 | 1 | The number of questions | 2 |
| Struct * | 1 | Pointer of questions buffers | 4 |
| Int 16 | 1 | The number of total HMM states | 2 |
| Int16 | 1 | The number of HMMs | 2 |
| Struct * | 1 | Pointer of HMM information buffers | 4 |
| Uint16 | 256 | Phoneme index | 2 * 256 = 512 |
| Struct * | 1 | Pointer of PCPinfor | 4 |
| 16-byte | Q | Questions | 16 * 326 = 5216 |
| Phoneme tree size | N | Phoneme data | 21136 |
| 10-byte | H | HMM information | 15664 |
| **N: number of phonemes**<br>**Q: number of questions**<br>**H: number of HMM states (Pel number)** | | | |

**Tab.1 The format of the swiarb file**

The swiarb file is created from external tools, it can be roughly divided into three groups. The first group is phoneme mapping that maps a phoneme from its phoneme code to an ASCII code. The second group is the question set; in these questions, underscore sign is used to indicate if a question is a left or right question. There are no “left AND right” questions in srec arb file. The third group is question binary tree. Answer “yes” will go to the right node, and answer “no” will go to the left node. “Question -1” means the node is a leaf node. A typical internal node looks as:

**node 0: question 166, left-> 1, right-> 10**
**genone 65535, pel 65535, dur mn -1, absdev -1**

It means that node 0 utilizes the 166$^{th}$ question; the answer “no” will track down to the node 10, and answer “yes” will track down to the node 1. Because the node 0 is not a leaf node, the question tree will not assign a Pel number, duration mean, and absolute deviation to it. A typical leaf node looks like:

**node 7: question -1, left-> -1, right-> -1**
**genone 12, pel 12, dur mn 70, absdev 45**

It means the node 7 is a leaf node, and there is no more questions. The triphone HMM mode will use the Pel 12; the state duration mean is 70 mille-seconds, and its absolute deviation is 45 mille-seconds.

Sources:
srec/clib/srec_arb.c

### 4.5 swimdl file format

The swimdl file is an acoustic model file, and it provides three kinds of acoustic model information, mixture components (they are usually Gaussian distributions), components mixture weight, and Pel durations.

Each component is usually a Gaussian distribution; several components are mixed together to build a Pel, and each component in a Pel has a mixture weight. Each Pel has a duration parameter, which define an average state duration time.

| Size unit | number | Description | Size of a typical srec gender specific acoustic model (Byte) |
|---|---|---|---|
| Short | 1 | number of Pels in the acoustic model | 2 |
| Short | 1 | number of dimensions in a component | 2 |
| Short | 1 | number of total components in the acoustic model | 2 |
| Short | P | number of components in each Pel | 2 * 1600 = 3200 |
| Char | M * D | all pdf mean values | 1 * 1600 * 6 * 36 = 345600 |
| Char | M | all pdf weight values | 1 * 1600 * 6 = 9600 |
| Char | P | Pel duration | 1600 |
| **P: the number of total Pels**<br>**M: the number of total Gaussian components**<br>**D: the feature dimension** | | | |

**Tab.2 The format of the swimdl file**

The swimdl file is a binary file. Tab.1 described the format of a swimdl file, and the column 4 is an example of a typical srec gender specific model; its total size is about 360k bytes.

Mixture weights were obtained from the Tracy mix file. In Tracy mix file, the formula of calculating weights is “-mul_scale * log(prob), where mul_scale = 6.4”. The SREC swimdl file also use the log value of the probability as the mixture weight, but the srec swimdl file does not apply minus mul_scale; therefore, before dumping weight values to the swimdl file, all weight values were divided by “-6.4”.

### 4.6 swimdllst file

The swimdllst file is a list of swimdl loaded in run time. Usually, there are two swimdl files in the list for gender-dependent acoustic models; one is male model, and the other is female model. The default silence model that the SREC engine loads is the one in the first model in the swimdllst file. Blank-lines are not supported in this file.

### 4.7 LDA file format

The lda file serves to improve the discriminant power the MFCC vectors coming out of the front-end. With front-end features represented by "x", the lda serves to apply a transformation, Ax+B, where A is a diagonalizing transform and B is a translation vector. The content of the .lda file is the inverse of A. We invert it at initialization time. Tab.3 described the format of a LDA file, and the size of a binary LDA file is 10954.

| Size unit | number | description | Size of a typical srec LDA (Byte) |
|---|---|---|---|
| Short | 1 | matrix dimension | 2 |

| | | | |
|---|---|---|---|
| Double | 1 | matrix scale | 8 |
| Double | D | eigenvalues | 288 |
| Double | D | translation vector | 288 |
| Double | D * D | inverse of matrix | 288 * 36 = 10368 |
| **D: LDA vector dimension** | | | |

**Tab.3 The format of the LDA file**

## 4.8 SR_AcousticState

This class represents encapsulates the acoustic state of the caller and calling environment during a call in order to improve recognition accuracy. The Load/Save/Reset functions of this class are intended to be used to enable channel adaptation to cross start-up/shut-down boundaries, but the functions are not fully implemented in version 1.0. Just before a shutdown, an application should call SR_AcousticStateSave() .. which saves the channel (cepstral mean) information to a file. At startup, if the acoustic state file exists, ithe SR_AcousticStateLoad() function should be used. Occasionally, the cepstral mean information may diverge due to very-high background noise, in those cases SR_AcousticStateReset() should be used to return to factory defaults.

Source code to see:
srec/AcousticState/src/AcousticState.c
srec/AcousticState/src/AcousticStateImpl.c
srec/clib/swicms.c
srec/include/swicms.h

# 5 SR_GRAMMAR

This class represents the recognition grammar, a collection of words and sentences the user may utter. Grammars may be loaded from or saved to disk.

The grammar compiler is based on cppdom library for parsing the input grxml. The grammar compiler generates fsm graph and .map files, which generally follow the AT&T FSM format documented here .. http://www.research.att.com/~fsmtools/fsm/man4/fsm.5.html

## 5.1 Text-format grammar files

The grxmlcompiler produces 5 main output files for each grammar. SR_GrammarLoad() adds the 6 file extensions when loading the 6-tuple.

| File | Input labels | Output labels | notes | Related Source code |
|---|---|---|---|---|
| .PCLG.txt | Hmm models | Words | Arc weights are allowed on this FST | CA_Syntax, `CA_LoadSyntax()`, `FST_LoadGraph()` |
| .Grev2.det.txt | Words | Words | This is an acceptor FST (i.e. an FSM); when the grammar is represented by "G", this is a rreversed then determinized version of that grammar. | `CA_Syntax, srec_context, CA_LoadSyntax(), FST_LoadReverseWordGraph()` |
| .P.txt | Words | Semantic scripts | This is an FST, also determinized to make the searching though quicker | `SR_SemanticGraph, SR_SemanticGraph_Load()` |

| File | Description | Related Source code |
|---|---|---|
| .map | Words, word orthographies | `srec/crec/srec_context.c, wordmap_*()` |
| .script | Tags used in the grxml grammar | `srec/crec/srec_context.c, wordmap_*()` |
| .params | Parameters used in the grxml grammar, this is used only for the word_penalty for now, but can be eventually be used to store other grammar specific parameters such as end-of-speech timeouts | `srec/crec/srec_context.c, FST_LoadParam_*()` |

The following additional constraints apply to each of these graphs for proper compatibility with the recognizer:

.PCLG.txt: (1) .wb input labels are used to denote word-boundaries. When cross-word acoustic models are used, the context-dependency is honored across the .wb arc. (2) All paths through the PCLG graph must have exactly 1 output label between .wb input labels, such that when the search encounters a .wb input label, the word completed is known. A side effect of minimization is often that output labels are pushed toward the front of the graph, but care must be taken to not push them past a .wb arc. This graph may or may not be fully minimized. (3) To support homonyms, multiple .wb input label arcs can leave the same node with different output labels, but if these carry different meanings, then care will need to be taken at confidence scoring time (score differences of 0 might not be handled properly). (4) slots: slots must be marked by 2 or 3 special arcs; there should be only a single reference to the slot output label (e.g. __Names__) because multiple references are not supported and would result in having to

add the items to both locations in the graph. For this reason, the 2 or 3 arcs are used to allow for merging on the sides.

.P.txt: the parser graph must be deterministic and arcs must be sorted such that epsilon input label arcs appear before any other arcs.

.Grev2.det.txt: the reverse word graph used in n-best decoding must begin with an epsilon and must be deterministic

## 5.2 G2G-format grammar file

The 5-tuple format of compiled grammar may not be convenient for deployment situations. The make_g2g tool is intended to be used for a more compact representation on the 5-tuple of files. The "g2g" grammar format can also pre-allocates space for possible slot-based word addition, such that the different components of the .g2g already have allocated space for their anticipated growth.

```
make_g2g –base mygrammar,addWords=100 –out mygrammar.g2g
```

| File | C-functions | Pre-allocations |
|---|---|---|
| .map | srec/crec/srec_context.c<br>wordmap_create() | space for extra characters and extra word (character pointers) is pre-allocated assuming a certain number of average number of characters per word. See AVG_CHARS_PER_WORD |
| .PCLG.txt | srec/crec/srec_context.c<br>FST_LoadGraph() | space for extra arcs and nodes is pre-allocated assuming AVG_ARCS_PER_WORD and AVG_NODES_PER_WORD |
| .script | Srec/crec/srec_context.c and<br>srec/SemProc/srec/SemanticGraphImpl.c<br>wordmap_create() | Space for extra scripts is pre-allocated assuming AVG_SCRIPTS_PER_WORD |
| .Grev2.det.txt | srec/crec/srec_context.c | space from the wordmap is re-used |
| .P.txt | srec/SemProc/srec/SemanticGraphImpl.c | space from the wordmap is re-used |

The pre-allocation makes it such that after a grammar is loaded, no new memory allocations are required during slot-based dynamic word addition, in order to keep the word–addition fast.

Note that .g2g files have weights (arc weights) built-into the lexical graph. Adding weighted entries to the grammar stores the weights on the arcs.

## 5.3 Slot-based word and voice-tag addition

Slot-based addition is supported by way a special words used in the grammar, such as "__Names__". At grammar compilation time, the pronunciation for those words is hacked such that 3 arcs are inserted in the PCLG.txt graph. At word-addition time, the PCLG.txt graph is search for these special arcs and the start-node and end-node for the slot are cached.

Even when the underlying acoustic models (used at grammar compilation and recognition-time) support cross-word context-dependency, slot-based addition does not honor it. At grammar compilation time, the carrier-phrase prefix is

assumed to be looking at silence to the right, and the carrier-phrase suffix is assumed to be looking at silence to the left. With regard to the word/voicetag being added it is assumed that silence is on each side, so again cross-word context-dependency is not honored.

Within the slot, context-dependency is honoured, but approximations are made when dealing with "multi-word" words. An example might be a contact name such as "First_Last". The pronunciation for this "word" might be "fVst&last" where but again, for cross-word context-dependent acoustic models, the cross-wording is not honoured. of the word/voicetag being added.

### 5.3.1 Dynamic word addition

The SREC engine has the ability to add words dynamically to a grammar. As a consequence, a user doesn't have to specify the maximum number of words that can be added to a grammar a priori, but instead he can set this maximum to zero (or strip it out), and the engine will dynamically allocate the memory space needed to add all of the words required. Of course, the user can still pre-allocate space during the grammar compilation time, however it's more recommended to add words dynamically to the grammar especially if their number is not known, thus saving some memory space.

In order to add words dynamically to a grammar, SREC checks if there are enough free arcs and node to be added to the PCLG.txt graph and if there are enough free characters space and words space that can accommodate the new characters and words the user wants to add to the existing ".map" and the ".script" characters and words spaces. If there aren't, SREC will expand the corresponding memory by 20%. If the result of the expansion is less than the minimum growth factor, then the memory will be expanded by this minimum too. Note that the minimum growth factor for arcs, nodes, characters and words is 100, 100, 256 and 32 respectively. After that, the engine copies the old information to the new memory, frees the old one and adjusts the related parameters and pointers. The table below shows all the parameters that SREC checks when adding a word dynamically to a grammar, the corresponding minimum growth factor, the pointers and parameters that are adjusted and the related source code:

| Parameter | Minimum growth factor | Pointers and parameters adjusted | Related source code |
|---|---|---|---|
| Arcs | 100 | - FSMarc_list : the arc list<br>- FSMarc_list_len : maximum number of arcs that can be added to the .PCLG.txt graph<br>- FSMarc_freelist : lists the available free arcs | srec/crec/srec_context.c :<br>fst_add_arcs() |
| Nodes | 100 | - FSMnode_list : the node list<br>- FSMnode_list_len: maximum number of nodes that can be added to the .PCLG.txt graph<br>-FSMnode_freelist: lists the available free nodes<br>-FSMnode_info_list : the node information list | srec/crec/srec_context.c :<br>fst_add_arcs() |
| Characters | 256 | - max_chars: defines the maximum number of characters that can be added to the ".map" and ".script" characters space.<br>- next_base_chars: points to the location where the next base character will be added.<br>- next_chars: next character pointer – points to the location where the next character will be added.<br>- chars: characters pointer – points to the first character added. | srec/crec/srec_context.c :<br>wordmap_add_word_in_rule()<br>wordmap_add_word() |
| Words | 32 | - Words: words pointer.<br>- max_words: defines the maximum number of words that can be added to the ".map" and ".script" words space.<br>- the hash table that saves mapping between wdID and array index | srec/crec/srec_context.c :<br>wordmap_add_word_in_rule()<br>wordmap_add_word() |

Finally, the SREC engines checks if there are enough free arc tokens for the scripts to be added. If there aren't, the same process stated above will be applied here. Also, the minimum growth factor for the arc tokens is equal to 100. Please check SR_SemanticGraph_AddWordToSlot() in the following C-file for more information: srec/Semproc/src/SemanticGraphImpl.c.

### 5.3.2 Resetting the slots

When the user decides to reset the slots of the grammar, the memory allocated for arcs, nodes, arc_token_list, characters and words will be freed, and all the pointers and parameters are set to their initial basic values (i.e. the values they had when the grammar is still empty). Note that after resetting the slots, the user will be offered dynamic word addition mode only, even if he had pre-allocated some space during grammar compilation time. In other words, the pre-allocated space will be lost once the grammar is reset. Please check SR_SemanticGraph_Reset() in the following C-file for more information: srec/Semproc/src/SemanticGraphImpl.c.

## 5.4 SR_SemProc

This module captures the functions related to semantic processing of recognition results. When a full nbest list is obtained, each entry in the list consists of a sequence of words (or wordIDs), representing the orthographic recognition result. The recognition result is used to trace a path through the parser graph (P.txt file). The sequence of arcs, a path, in the parser graph whose input labels correspond to the recognition result is captured. A recursive function is used to traverse the graph, such that one path through the graph is returned. Multiple paths are not searched, only the first one is used. The sequence of arcs is re-scanned for output labels, corresponding to script-ids and scope-ids such that they can be accumulated into a mini-program.
Srec/SemProc/src/SemanticProcessorImpl.c can be configured (see “Accumulated Scripts for”) to dump the list of script-ids, and hence the actual scripts to be executed. The scripts are simply executed in forward order, with the final values of all variables available to the API. In some cases, variables that have lost scope are not cleared.

In theory, a developer may change the <tag> scripts in the grammar and use a script interpreter other than the “eScript” interpreter provided in srec/SemProc/src/LexicalAnalyzer.c ExpressionEvaluator.c ExpressionParser.c .. to support more complex or more standards-oriented tags (such as ECMAscript).

Documentation on the “eScript” language is provided in elsewhere.

# 6 VOICETAGS: SR_NAMETAG, SR_NAMETAGS

SR_Nametag represents phonetic Nametags, used for voice-enrollment. A nametag is created from a recognition result and may be inserted into dynamic grammar slots for subsequent recognitions.

SR_Nametags represents a collection of nametags and may be loaded from or saved to disk. This class is not used by the UAPI.

Utterances for voicetags are decoded using the regular speech recognizer; with the application is expected to load the enrollment grammar prior to recognition. The recognition result from that grammar can be decoded into a transcription, which can then be use in slot-based voicetag addition. Two types of enrollment grammars are supported in the code (see `USE_ENROLL_HMM_BASED`), but in practice only 1 is needed. Hmm-based enrollment results in slightly better accuracy, but the decoded transcription does not have a phonological meaning as it does in phonetic-based enrollment.

## 6.1 Phonetic-based enrollment

When the enrollment grammar is any-phoneme followed by any-phoneme, then the recognition result is a sequence of phonemes. The decoded meaning might look like this:

ph_a ph_b ph_a ph_k ph_u ph_s

This represents a phoneme sequence; when this sequence is passed into the SR_GrammarAddNametagToSlot() function the context-dependency inside this voicetag will be honoured.

When the recognizer is decoding the phoneme sequence, the source code in srec/Recognizer/src/RecognizerImpl.c is hacked to disable nbest re-ordering. Nbest re-ordering described further below, has the effect of occasionally displacing the top choice with an alternate choice of higher score (lower cost). However, the alternate choice does not honor cross-word context-dependency due to approximations in nbest decoding. As such, the true top choice from the forward pass decoding should be used, so that the hmm sequence at word addition (which does honor internal context-dependency) ends up being the same as the hmm sequence decoded with the enrollment grammar.

The enrollment grammar is generally located in the esr/config/en.us/grammars/enroll.grxml file. Only generic phonemes are

## 6.2 Hmm-Based enrollment

When the enrollment grammar is a any-hmm followed by any-hmm, then the recognition result is a sequence of hmms. The decoded “meaning” might look like this:

wd_hmm369_e wd_hmm525_n wd_hmm330_d wd_hmm588_r wd_hmm761_} wd_hmm384_f wd_hmm573_p

This represents an hmm sequence, which does not necessarily honor context-dependency, but which can non-the-less be passed into `SR_GrammarAddNametagToSlot()`which will handle it appropriately. Note that only the hmm numbers are retained in this case, the phoneme tags at the end are used only for sanity checking. It is highly discouraged for developers to interpreting the phoneme sequence for some other usage because context-dependency may change the hmm sequence in important ways that make the voicetag less accurate.

# 7 SR_VOCABULARY & G2P ENGINE

A vocabulary maps words to their phonetic representation. This mapping it backed up by a lookup table (loaded from disk) or a TTP engine for unknown entries. Vocabularies are language-dependant and adaptive; they may be updated with new entries and saved to disk.

## 7.1 Dictionary

The user dictionary in SREC is located near esr/config/en.us/dictionary/large.ok as referenced by the cmdline.vocabulary option in the baseline.par fle. It has the first priority for phrase lookups.

If a phrase is not found there, then the phrase is split up into words, and words are looked up independently. Words for which no pronunciation is found are sent to the G2P engine.

At this time, the total dictionary includes 29870 words. There are two columns in the dictionary file. The first column is the word spelling, and the spelling is case-sensitive. The second column is the phoneme pronunciation represented by ASCII code; each code represents a phoneme. The phoneme list and its correspond ASCII representation is in the phonology chapter of the SREC User Guide.

SREC always uses the dictionary referenced from the baseline.par file cmdline.vocabulary. This setting can be changed however, such that at grammar compilation time a very large dictionary is used, but at run-time on the target device, as shorter dictionary is used according to memory constraints.

## 7.2 G2P Engine

The G2P engine is triggered when a word lookup in the dictationary fails. The letter to phoneme conversion operates using a context dependent decision tree with questions associated with letters to the right and left, phonemes to the left (see QuestionType near seti/sltsEngine/include/lts_seq_internal.h). The data file for the G2P engine is located at config/en.us/g2p/en-US-ttp.data. Tools to generate this file are Nuance proprietary.

For historical reasons, the G2P engine generates phonemes in a format called “ETI-Eloquence”. A conversion to the “SREC” format is applied after that.

The G2P engine does not apply any normalization, non-alphanumeric characters (other than quote “”) are replaced by a space, so effectively ignored.

## 7.3 Phonetic Inventory

The phonetic inventory for SREC is specified in the esr/config/en.us/models/generic.pht file.

# 8 SR_RECOGNIZER

The speech recognizer binds SR_AcousticModels and SR_Grammars and uses them for recognition. The recognizer takes in audio samples, processes them and returns recognition results. In case of successful recognition, the results contain a list of semantic results (multiple semantic results per nbest-list entry).

## 8.1 The main search

The SREC search is based on a token passing algorithm, with the active search "states" carried from frame to frame on "tokens". The graph being searched is the PCLG.txt file that is generated by the grammar compiler, possibly with words added after the fact. In this context, active search "states" refer to the cumulative score (we actually use costs) up to a particular point in the search graph, having added all acoustic and LM scores up to this frame. There are two kinds of tokens, fsmnode_token and fsmarc_token. The fsmnode_token sits on a node in the graph, and carry the the cumulative cost and the word history (plus alternatives). The fsmarc_token sites on a an arc of the graph, and is similar to a fsmnode_token, except that the cost and word history are maintained for each HMM-state of the arc (most of the acoustic models have 2 HMM states).

A fixed number of fsmnode_token's (and fsmarc_token's) are allocated at recognizer initialization time (srec_ allocate_recognition1()). At utterance initialization time (srec_begin()), the fsmnode_token's (and fsmarc_token's) are arranged in a linked list of free tokens (aka the freelist). During the search, new tokens that are needed are simply taken from from the freelist and put into the active list. This way, no memory needs to be allocated or free during the decoding.

**** Initialization ****
The search begins by initialization of a new fsmnode_token at the start node of the graph (0). The cost there is 0 and the word history is blank.

**** Propagation ****
The search is propogated from frame to frame (including from the first frame) as follows (also see code comments for more detail):

1. compute model scores. We scan through all active fsmarc_token's and fsmnode_token's, and determine the full list of acoustic models which need to be scored, then calls the actual scoring functions.
2. handle all internal fsmarc_token updates based on new frame observations. For a 2-state HMM (states 0 and state 1), this means we update a state based on the best history to propagate. State 'n' may have come a transition from from State 'n-1' or from a loop on State 'n'. (transitions into state 0 are handled in #2.
3. for each fsmnode_token (from previous frame), update into state 0 of the fsmarc_token sitting on to every arc leaving this node. When there is no fsmarc_token sitting on the arc, then we create a new one. When this process is done, no fsmnode_tokens remain, the entire search state is represented on fsmarc_tokens.
4. we prune fsmarc_tokens, those with scores too poor (costs too high) are deleted.
5. reset best cost to 0, to keep costs in a 16 bit range
6. for each fsmarc_token, we create a fsmnode_token for the node this arc goes to, if the fsmnode_token already exists then we simply update it (and merge in the path)
7. "update epsilons". The "eps" and ".wb" input labels on the FSM consume no input frames, so the costs are simply propogated for free (at least acoustically free). The .wb input label is special, it indicates that we are at the end of a word, so we must drop the current word to the word lattice and make it part of the word history.

Each looping over these steps consumes one frame of input, and advances the search by that much. The side effect of each loop is to possibly drop off some word_token's into a word lattice which encodes the word history.

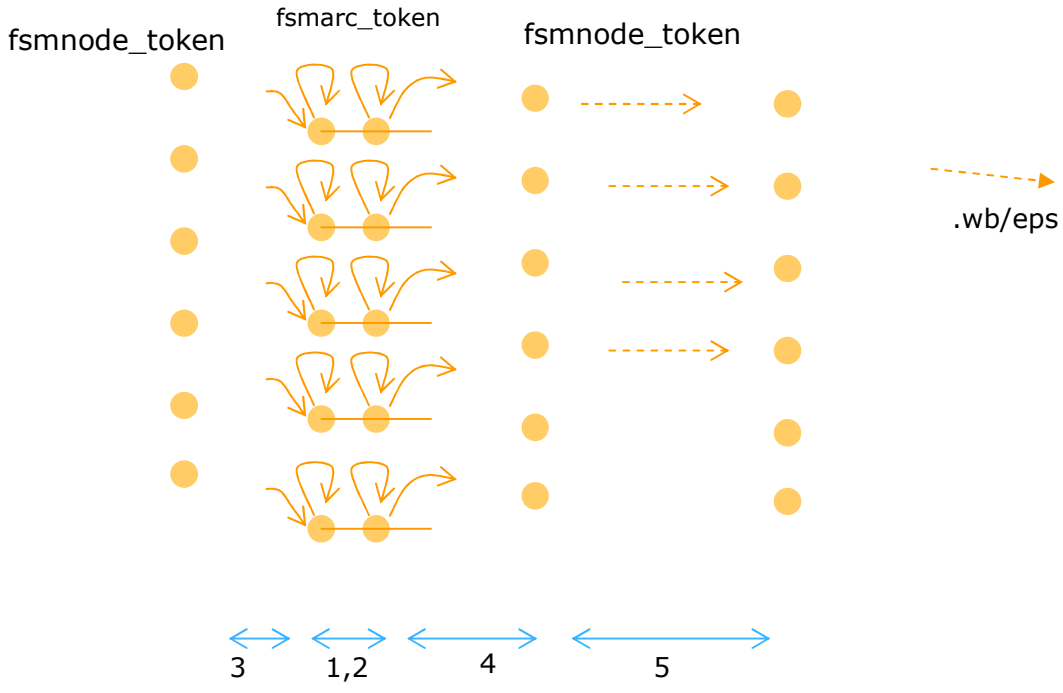
**** Word History ****
Word history is maintained using linked lists of word_tokens. Each word_token points to the previous word in the history, so at any point in the search we can always tell what the word sequence recognized so far is (for a particular

fsmnode_token or fsmarc_token). To get the best overall word sequence, we simply need to find the best fsmnode_token and trace that one back through the word_token's.

At recognizer initialization time, a fixed number ("CREC.Recognizer.max_word_tokens") of word_token's are allocated. At utterance initialization time, these are organized in a freelist and new ones are grabbed from the freelist as needed. Whenever a ".wb" arc is encountered during the search, a new word_token is generated at this frame by grabbing a word_token from the the freelist and placing it in the word_lattice. The word_lattice structure stores that list of word_tokens in a linked list, so one linked list for every frame since the beginning of the search. CREC.Recognizer.num_wordends_per_frame" indicates the number of word_tokens that are allowed to exist at any given frame, corresponding to words ending at that frame. When SREC runs out of word_tokens, it performs a repruning, which is to say, we propogate back all active histories and delete any histories that have not survived to this point.

The search is ended when the end-of-media is encountered (eg. end of file for file based input) or when one of the utterance timeouts are encountered (see timeouts elsewhere in this document). The two cases are handled the same way, SREC kills all active paths (fsmnode_token's) not on the final search node, then drops the active word (the "-pau2-" ending silence word) to the word_lattice.

The top choice transcription is easy to trace back. The top scoring word_token at the last frame can be traced back through the word_lattice.



## 8.2 Support for multiple acoustic models

SREC runs two acoustic models in parallel, one male and one female. The search is organized in such a way that these two search operate completely independently except for:

- the scoring of the silence model
- the overall best score

Silence scoring is shared so as to make sure no cross-pruning takes place on the basis of silence. The two acoustic models *should* have the same silence model, but the engine code guarantees that only one of the silence models is used.
The overall best score is shared, so that cross-pruning can take effect. This causes the mismatched gender generally to prune away more fsmarc_token's and fsmnode_tokens's. When no tokens exist for a this search, then it is terminated allowing the other to proceed unhindered.

## 8.3 End of the search and end of utterance timeouts

The search ends if there is no more input media expected and none more to consume. The search can also end if one of the timeouts is exceeded. SREC uses grammar-based end of speech (EOS) detection and audio-level-based EOS detection as insurance.

The audio-level-based EOS waits for a specified number ("cmdline.silence_duration_in_frames") of consecutive low-energy frames to occur. In some applications we don't want EOS to be detected too early. The "cmdline.end_of_utterance_hold_off_in_frames" parameter is used to declare that hold off. Regardless of the where the best oath in the grammar is at that point, the search will end. Top choice and nbest choices will be determined, but if there were no active paths on the end node, then the recognition will fail.

The grammar-based EOS detection works by looking at best-scoring active path, represented by an fsmnode_token sitting on the end node. When the best-scoring active path continues to be that fsmnode_token on the end node for a specified number of frames ("CREC.Recognizer.terminal_timeout") then the search is ended.

An optional end node is a node that is only epsilons and/or silences away from the end node, but also loops back to nodes earlier in the grammar. An example of this is a grammar involving variable length digit recognition. It is generally desirable to use a longer timeout so we can be sure the user has finished speaking and is not merely pausing, such as between groups of digits. This is implemented by designating certain nodes in the grammar as optional end nodes, and if the path ending at this node continues to be the best scoring path, then SREC waits for a specified number of frames ("CREC.Recognizer.optional_terminal_timeout") before ending the search. Optional end node designation overrides the end node designation because scores n the optional end node and the graph end node will generally be the same when they are separated only by epsilons and silences apart.

An non-terminal node is neither optional end node nor end node. In some cases the background noise is so high that search prefers to remain looping on the final "s" of the digit six, rather than loop on silence. When the best path continues to correspond to that node for a specified number of frames ("CREC.Recognizer.non_terminal_timeout") then we end the search.

All of the method above rely on the best path continuing to correspond to the same node. This is further strengthened by requiring that this best path be best by a certain margin ("CREC.Recognizer.eou_threshold").

## 8.4 The nbest search

When the end of utterance has been detected or forced, the nbest search is executed. The nbest list is generated using an a-star algorithm, which traces paths backwards through the reverse word graph corresponding to the grammar. The word_token identifies the word ending at this frame, and holds a word_backtrace pointer which indicates what word precedes it. The preceding word ends at a particular frame, but other words ending at that frame are stored in the word lattice, and can be considered as alternatives. This list of alternative words is further reduced by intersecting it with the list of words that are grammatically possible at that point, per the reverse word

graph. Note that this gives only an approximate nbest, since the actual joining frame for alternate histories may be different from the joining frame for the top history.

## 8.5 Confidence value calculation

SREC computes confidence values for the top choice of each recognition result. This value should not be confused with the "raw" recognition score, which is the log-likelihood of each nbest entry. The log-likelihood raw scores are costs, the higher the cost, the less likely this nbest entry hypothesis is the truth. Those scores have little meaning outside of SREC.

The confidence values do have meaning, ranging between 0 and 100. This value is intended to be used for comparison with a predefined threshold, such that if the confidence value is larger then the recognition result should be accepted, but when the confidence is lower then the recognition result should be rejected. A second threshold can be also be used to create a confirmation range. This logic is expected to be coded at the application layer.

Confidence scoring takes place after semantic interpretation, described in the SR_Grammar module. This is important since the confidence scoring method depends among other factors, on the raw score difference between the first and second choice. When that differences are not semantically important (eg. "zero" versus "oh", or "call" versus "dial"), then this help improve confidence values.

The confidence value is calculated from measurements taken on the nbest list. These are:

- Score-difference12 (between the top and second choice)
- Score-difference13 (between the top and third choice)
- Absolute-score (raw score)
- Score-per-frame raw score divided by number of frames that contributed to that score)
- GSM-score-difference (score difference between the top and GSM-score)

These measurements are each passed through a sigmoid function, so as to normalize them to be between 0 and 1, with the angle and offset of the sigmoids determined empirically from test data. A geometric weighted average of the sigmoid outputs is calculated, and normalized to the 0-100 range to get the final confidence value. When there is only 1 choice on the nbest list then a subset of the nbest measurements are used, with weights and offsets specific to the 1-choice length lists. The weights and offsets of this model are specified in the "rejfile" that the baseline.par file points to (shared/sigmoid_param).

The GSM-score-difference: GSM stands for general speech model. The goal is to calculate a true probability of the each hypothesis, H, given the input utterance, U. So,

$$P(H|U) = P(U|H)P(H) / P(U)$$

The raw-scores described above correspond to the product of acoustic and language models scores, P(U|H) and P(H), respectively. Since P(U) is the same for all different hypotheses, H, this is generally excluded when sorting nbest lists, and is used only for confidence value calculation. In SREC, the P(U) is calculated as the product of best acoustic likelihood at every frame, corresponding to a general speech model in which any state and be followed by any other state. In some systems, extra states other than the subset needed by the active recognition grammar are scored, but since SREC is generally CPU constrained, only the active states being scored contribute to this "general speech model".

# 9 GRAMMAR COMPILER

The grammar compiler is used to compile grammars from grxml to AT&T FST text format FST files which SREC can use. Please refer to the grammar section above for a description of SRGS compliance and the grammar compiler outputs.

TinyXML available from http://sourceforge.net/projects/tinyxml is used to load the xml document into a recursive node structure. A parseNode() function is used recursively to parse each node in the document and build up graphs and sub graphs. The top level graph which is built up is the parser graph, "P", a transducer which maps input words to output tags. This is dumped as the $base.P.txt file, with associated input labels $base.map and output labels $base.omap. The input labels are words in the grammar and the output labels are semantic tag IDs or scope markers. This step operates completely independently of the SREC run-time engine, although certain constraints associated with the engine are documented in the grammar description above must be adhered to.

**$base.grxml → $base.P.txt $base.map $base.omap $base.script**

There are two implementations for the rest of grammar compiler, one with cross-word context dependent acoustic models using the OpenFST library (http://www.openfst.org) and one using within-word acoustic models. The default build uses the OpenFST library. This library can also be used to replace many of the graph functions used to build the parser transducer but this has not been done yet. When that is done the code associated to within-word model grammar compilation can also be cleaned out.

The "G" or grammar transducer is generated by projecting the input labels of "P", i.e. delete the output tag labels. Some OpenFST graph operations are performed to get the reverse determized "G" FSM which SREC uses.

**G = projinput(P); srecG = cat( min( det( rme( rev( G)))), eps)**

… where projinput() refers to fst projection of the input labels; cat() is catenation, min() is minimization, det() determinziation, rme() is remove epsilons, and rev() is reverse.

The SR_Vocabulary() functions are used to build up an "L" transducer from the $base.map file. Each word in the grammar is looked in the dictionary, or it's pronunciation is derived from the SETI engine. The "L" transducer has phonetic pronunciations for input labels (see esr/config/en.us/models/phones.map) and words for output labels. Special considerations are given for homonyms.

The "C" transducer maps acoustic models for input labels into phonemes for output labels. It is built with the make_cfst tool provided with the SREC toolkit. The generic "C" transducer comes pre-bullt in the SREC SDK, but can be re-built with the following command line:

```
# cd esr/config/en.us/models
# make_cfst –swiarb generic.swiarb –cfst generic.C
```

To build the $base.PCLG.txt file the grammar compiler basically runs:

**SREC-PCLG = min( C * det( rme( L * cat( pau, G, pau2))))**

.. where "pau" and "pau2" refer to the prefix silence and suffix silence FSTs, which are built on the fly at each compilations. Special considerations are made for :

- homonyms … we add extra pseudo-epsilons before determinations
- slots … we use 3 arcs and a custom "pronunciation" to encourage merging paths into a single start of slot node and a single end of slot node.
- min() .. before the minimize operation, we merge input and output labels so that the output labels are not pushed earlier than .wb (word-boundary markers)

The SREC-PCLG is written to the $base.PCLG.txt file, whose input labels are now acoustic models (esr/config/en.us/models/models128x.map) and whose output labels are words in the grammar ($base.map).

The .params file is used store any parameters specific to this grammar. At this time, only the word_penalty parameter is stored here, but this can be extended to include post-speech timeouts and other grammar specific parameters.

# 10 CONTRIBUTORS

The following is a non-exhaustive list of contributors to the SREC engine, with their affiliation as of their contribution:

| Name | Contributions | Contact |
|---|---|---|
| Jean Dahan, Nuance Communciations | Speech science, core engine development | jean.dahan@nuance.com; jean.dahan@gmail.com |
| Mike Phillips, Nuance Communications | Core search, G2P engine | |
| Johan Schalkwyk, Nuance Communications | Core search design | |
| Gili Tzabari, Independent contractor | SR_ API, SR_ internals | gili.tzabari@bbs.darktech.org |
| Rabih Majzoub, Independent contractor | Confidence scoring | rabih.majzoub@nuance.com |
| Xufang Zhao, Independent contractor | Acoustic models | xufang.zhao@nuance.com |
| Tadashi Yonezaki, Zhihong Hu, Alexander Barbosa, Nuance Communications | Acoustic models | |
| Alex DeMarco, Independent contractor | Semantic interpretation | |
| Maha Kadirkamanathan, InfinitiveSpeech Systems, PatMach consulting | Grammar compiler | mahakadir@yahoo.co.uk |
| Various scientists at Dragon Systems circa 1994 | Front-end | |
| Yun-Sun Kang, Nuance Communications | G2P engine | |
| Dennis Velasco, Nuance Communciations | QA, feature development | dennis.velasco@nuance.com |
| Steve Raineri, Nuance Communciations | QA, feature development | steve.rainieri@nuance.com |
| Andy Wyatt, Nuance Communciations | Project Management | andy.wyatt@nuance.com |