SVOX AG
Baslerstrasse 30
CH-8048 Zürich

phone +41 43 544 06 00
fax +41 43 544 06 01
www.svox.com

SVOX Pico

Lingware Tools and Source File Description

1.0.0

May 20th, 2009

Table of Contents

1 Introduction

In this document, we report information useful to rebuild the individual lingware components from parameter (source) files. The lingware used by the Pico engine is composed of a collection of ‘resources’ that, in their binary form, are conventionally named ‘pkb’, i.e. (p)ico (k)nowledge (b)ase. The terms ‘knowledge base’ and ‘pkb’ are then to be considered synonyms in this document.

The actual lingware used by the runtime system is a suitably-packed binary collection of pkbs. More on how this packing is performed can be found on the document "SVOX Pico Core System, Software Architecture and System Development Guidelines Version 1.6".

In this description, we will detail how the individual pkbs can be generated starting from a ‘lingware source file’ or ‘lingware parameter file’ representation.

1.1 The Lingware Build Process

In general, the lingware build process can be split into the following steps:

1.1.1. Preparing the lingware source files
1.1.2. Generating the individual pkbs using the provided tools
1.1.3. Packing the pkbs on the final lingware for the runtime using the provided tools

Most of the details we give in this description are related to the step 1.1.1 and 1.1.2 (i.e. what is the content of the lingware sources and how to generate the pkbs). The step 1.1.3 is just a matter of invoking a script and is not described here.

1.2 Pkb List

This is the list of individual resources needed to generate a complete set of lingware files:

1.2.1.	TPP_MAIN	text pre-processing knowledge base
1.2.2.	TAB_GRAPHS	graphemes accepted in the language
1.2.3.	TAB_PHONES	phonemes accepted in the language
1.2.4.	TAB_POS	allowed PoS classifications
1.2.5.	LEX_MAIN	main lexicon
1.2.6.	DT_POSP	decision tree for PoS prediction
1.2.7.	DT_POSD	decision tree for PoS disambiguation
1.2.8.	DT_G2P	decision tree for G2P disambiguation
1.2.9.	DT_PHR	decision tree for phrasing prediction
1.2.10.	DT_ACC	decision tree for prominence prediction
1.2.11.	DT_DUR	decision tree for duration prediction
1.2.12.	DT_LFZ1..5	decision tree for pitch prediction
1.2.13.	DT_MGC1..5	decision tree for cepstral prediction
1.2.14.	FST_WPHO_1..5	FST for sentence phonology
1.2.15.	FST_SPHO_1..5	FST for syllabification
1.2.16.	FST_XSPA_PARSE	FST for XSAMPA parsing
1.2.17.	FST_XS2SVPA	FST for XSAMPA to SVOX-PA mapping
1.2.18.	PDF_DUR	PDF tables for duration
1.2.19.	PDF_LFZ	PDF tables for pitch
1.2.20.	PDF_MGC	PDF tables for cepstrum and source
1.2.21.	PDF_PHA	PDF tables for phase

Legend

PoS:	part of speech
G2P:	grapheme to phoneme conversion
DT:	decision tree
FST:	finite-state transducer
XSAMPA:	extended SAMPA phonetic alphabet
PDF:	probability density function

2 Description of FST Source Files

Applies to 1.2.14 ... 1.2.17

2.1 Introduction

FSTs are specialized finite-state automata in which the transition 'symbols' are actually pairs of input and corresponding output symbols. In order to transduce an input string to a corresponding output string, a state transition path must be found which ends in an accepting state and where the input symbol of each transition corresponds to a symbol of the input string. The output symbols of the found transition path together form the output string. Empty symbols are allowed on both input and output side.

The current FSTs are deterministic automata with respect to the symbol pairs, that is, for each symbol pair there is at most one transition with that pair from any state to another state. However, with respect to only the input symbols the FST is not deterministic, i.e., an input string may, in general, yield several output strings. (However, this should be avoided in the FSTs used in the Pico system.)

The FST source files consist of three sections: the alphabet definition (symbol pair set), the grouping of pairs into classes, and the state-transition matrix. The pair classes and the transition matrix together form the actual automaton.

2.2 Alphabet

Syntax:

```
Alphabet = ":ALPHABET" {Pair} "*" .

Pair     = Symbol [ "/" Symbol] .

Symbol   = "'" {<any character with doubling of '>} "'" |
           '"' {<any character with doubling of ">} '"' |
           "@" .
```

The alphabet starts with the keyword `:ALPHABET`, is followed by a list of symbol pairs, and ends with '`*`'. Symbol pairs with identical input and output symbol can be denoted by only a single symbol, pairs with different input and output symbols must note both symbols separated by '`/`'. The individual symbols must be strings delimited by either single or double quotes. If the delimiter character must appear in the string, it must be doubled (e.g., `""""` is the string containing exactly one double quote). The null symbol (epsilon symbol) is represented by `@` (not quoted!). The pair `@/@` or only `@` is not allowed, that is, the null symbol must always be either input our output symbol but not both.

Example:

```
:ALPHABET 'a' 'b' 'c' 'd' '*' 'a'/'u' 'b'/'v' 'c'/'w' 'd'/@

*
```


2.3 Classification of Pairs

Syntax:

```
Classification = ":DEFAULTCLASS" Number {":CLASS" Number ":IS" {Pair}} .

Number         = Digit {Digit} .
```

In order to keep the transition matrix of the FST small, the symbol pairs are grouped into equivalence classes. All symbol pairs of such a class have identical transitions. Each symbol pair logically occurs in exactly one class. Pairs that are not explicitly listed in a class are put into a default class, the number of which must be declared as well. The classes are numbered from 1 to <nrClasses>. The numbering need not be ordered.

Example:

```
:DEFAULTCLASS 2

:CLASS 1 :IS  'a'/'u' 'b'/'v' 'c'/'w' 'd'/@

:CLASS 3 :IS  '*'

.
```

2.4 Transition Matrix

Syntax:

```
TransitionMatrix = ":TRANS" StateNumber ":TO" {StateNumber} .

StateNumber      = Digit {Digit} .
```

The transition matrix notes the full matrix of transitions from all starting states with all pair classes to ending states. The first state of the automaton is state 1. Each `:TRANS` row must have exactly <nrClasses> target states. If a transition does not exist, the target state is noted as 0. The starting states (after `:TRANS`) must be in strictly ascending order from 1 to <nrStates>. In the current framework there is only a single accepting state, namely state 1. (Every automaton can be brought into this form by means of a special additional terminator symbol which leads from all previously accepting states to a newly introduced single accepting state which must be numbered as state 1.)

Example:

```
:TRANS 1 TO  0  1  2  [ending states for classes 1, 2, and 3, starting
state 1] .

:TRANS 2 TO  1  0  2  [ending states for classes 1, 2, and 3, starting
state 2] .
```

2.5 Full FST

Syntax:

```
FST = Alphabet ":AUTOMATON" Name Classification TransitionMatrix "*" .

Name = '"' {<any character with doubling of double quotes>} '"' .
```


The full automaton consists of the alphabet, the classification section and the transition matrix. The classification is preceded by an optional automaton name after the keyword `:AUTOMATON`. The automaton is closed by a final "`*`".

Example:

```
:ALPHABET 'a' 'b' 'c' 'd' '*' 'a'/'u' 'b'/'v' 'c'/'w' 'd'/@
*
:AUTOMATON "SUBS"
:DEFAULTCLASS 2
:CLASS 1 :IS  'a'/'u' 'b'/'v' 'c'/'w' 'd'/@
:CLASS 3 :IS  '*'
:TRANS 1 TO  0  1  2
:TRANS 2 TO  1  0  2
*
```

This automaton describes an FST which accepts 'a', 'b', 'c', 'd', and '*' as input symbols. After each '*' in the input, any immediately following 'a', 'b', 'c', or 'd' is converted into 'u', 'v', 'w', and null, respectively.

2.6 Comments

There are two forms of comments:

- Line comments are denoted by ‘!’, that is, any text following a '!' on the same line is ignored.
- Range comments are denoted by ‘[‘ and ‘]’, that is, any text between square brackets is ignored. Range comments may be nested.

3 Description of DT Source Files

Applies to 1.2.6 ... 1.2.13

3.1 Introduction

The DT (Decision Trees) based resources are built using the tool ‘dt2pkb’, operating on an input textual file whose content is described in the following.

3.2 dt2pkb Tool Invocation

The `dt2pkb` tool compresses the decision tree (DT) and builds its bit stream representation (a binary file with the pkb extension). The tool takes three arguments:

```
<dtfmt_file> <dt_file> <output_pkb>
```

Example:

```
dt2pkb lex_en-GB_variant2.amb.dtfmt lex_en-GB_variant2.amb.DT.utf lex_en-GB_variant2.amb.DT.pkb
```

Where:

`<dtfmt_file>` : configuration file

`<dt_file>` : input DT file

`<output_pkb>` : output pkb file

3.3 Configuration File Description

`<dtfmt_file>` : the configuration file contains the definitions of the data types used in the vector attributes and in the DT questions:

`:ATTR "<value>"` (ex.: `:ATTR "1"`) : the order number of the vector attribute and the question in the DT. The values are in the range [1, 255] and the reserved word ‘target’ is available. The word ‘target’ introduces the data type for the decision values.

`:PROP [table = "<path>"] type = <value>` (ex.: `:PROP table = "..\phoneset.table" type = phone`) : the properties defining the data type of the current attribute.

`table = "<path>"` is an optional property, which defines the path to the data table containing the full set of the available values with their indexes.

`type = <value>` is the data type. The following types are defined:

- phone: a phoneme value;
- graph: a grapheme value (the UTF-8 characters of the language alphabet);
- pos: a part of speech tag;
- history: an index of the output decision;
- numeric: an integer value.

Example:

```
:ATTR "16" :PROP type = history !the index value
```



```
:ATTR "target" :PROP table = "..\..\..\tables\de-DE\phoneset.table" type
= phone
```

Comments are supported in the <dtfmt_file> and are introduced by '!' (exclamation mark).

3.4 Decision Tree File Description

<dt_file> : the DT in text format (UTF-8 encoding).

One line in the file corresponds to one tree node:

```
<header>
<node_1>
<node_2>
.......
<node_n>
```

`<header>` : the header defines the DT parameters.

`size = <int>` : the size of the DT in terms of the number of the nodes;

`num_attributes = <int>` : the size of the input vector, which is equal to the number of the different questions asked in the tree; the parameter is optional.

`<node_i>` : there are four types of nodes in the tree

a) binary node;
b) continuous node;
c) subset node;
d) decision node.

The subset nodes have unlimited number of children (n outcome forks). The decision nodes are terminals and have no questions to ask. All other types of nodes have only two yes/no children.

`node_num = <int>` : the index of the node. The nodes index counting starts from 1 (`node_num = 1` is the root of the DT).

`quest_num = <int>` : the index of the question (it starts from 0), which is equal (i+1) to the order number of the vector attribute.

`subset = <val_1,val_2,..,val_n>` : the subset question determines the subset of the values for the corresponding vector attribute. A comma is the only possible delimiter between the values (no space, no tabulator, etc.)

`is_true` : the binary question.

`is_greater = <int>` : the continuous question determines a threshold.

`node_yes = <int>` : the index of the child node, if the answer is positive.

`node_no = <int>` : the index of the child node, if the answer is negative.

`decision = <val>` : the predicted decision value.

Examples of the nodes:

```
node_num = 29 quest_num = 12 is_true node_yes = 73 node_no = 72 !binary
node

node_num = 90 quest_num = 11 is_greater = 123 node_yes = 182 node_no =
181 !continuous node

node_num = 2 quest_num = 4 subset = e,a_LONG,y node_yes = 4 subset =
l,r,u node_yes = 5 node_no = 6  !subset node

node_num = 83 decision = _epsilon_ !decision node
```

Comments starting with the '!' are supported in the <dt_file>.

4 Description of Property Table Source Files

Applies to 1.2.2 ... 1.2.4

4.1 Common Syntax

Several lingware source files are simple tables which map symbols to associated properties. They share the following (incomplete) syntax:

```
PropertyTableFile = { Entry } .

Entry =  ":SYM" Symbol  ":PROP" Property { "," Property } .
```

The symbols introduced by the `:SYM` keyword must be strings delimited by either single or double quotes. If the delimiter character must appear in the string, it must be doubled (e.g., `""""` is the string containing exactly one double quote).

Two forms of comments are accepted:

- Line comments are denoted by '`!`', that is, any text following a '`!`' on the same line is ignored.
- Range comments are denoted by '`[`' and '`]`', that is, any text between square brackets is ignored. Range comments may be nested.

The legal strings of `Symbol` and the syntax and legal values of `Property` are left open to the different tables described in the following sections.

4.2 Phones Table

The phones table (`<LANG>_phones.utf`) defines the valid set of symbols together with properties needed during analysis and synthesis.

The phone table syntax adheres to the general Property Table syntax (section 4.1):

```
PhonesPropertyTableFile = { Entry } .

Entry =  ":SYM" Symbol  ":PROP" Property { "," Property } .
```

Properties are defined after the symbol and introduced by the keyword `:PROP`:

```
Property = "mapval" "="  INT
           | BoolProp "=" "1" .
```

Here is the complete list of the properties supported in the phones table. Notice that all properties with the exception of `mapval` (i.e. `BoolProp`) accept "1" as the only possible value.

- `mapval`: the property `mapval` is mandatory and defines the integer value (<=255) used internally. This value is unique and constant; if the value for any symbol in the table is changed, new lingware should be regenerated accordingly.
- `vowel`: set to 1 if the symbol represents a vowel.

- `diphth`: set to 1 if the symbol represents a diphthong.
- `glott`: set to 1 if the symbol represents a glottal stop.
- `nonsyllvowel`: set to 1 if the symbol represents a non-syllabic vowel, used in the text analysis as an alternative way of transcribing diphthongs.
- `syllcons`: set to 1 if the symbol represents a syllabic consonant.

The following properties may occur once only:

- `primstress`: set to 1 if the symbol is the primary stress.
- `secstress`: set to 1 if the symbol is the secondary stress.
- `syllbound`: set to 1 if the symbol is the syllable boundary.
- `wordbound`: set to 1 if the symbol is the word boundary.
- `pause`: set to 1 if the symbol is the pause.

4.3 Part-Of-Speech (PoS) Table

The table with the part-of-speech-tags (LANG_pos.utf) defines the valid set of PoS symbols used in the Pico system.

The Part-Of-Speech tag table syntax adheres to the general Property Table syntax (section 4.1):

```
PoSPropertyTableFile = { Entry } .

Entry =  ":SYM" Symbol  ":PROP" Property { "," Property } .
```

Two different types of PoS tags are possible: simple and complex tags. Complex tags are constructed out of the single ones separating every part by the caret symbol “^”, for example `ADJ_PL^V_PL`. Complex tags have introduced in Pico to deal with the ambiguity found in natural languages, like in the case of “I like your *accent*” and “I had to *accent* it” (`N^V`), where a different transcription should be predicted for each case.

When building the Pico lexicon it is important not to introduce complex PoS tags not available the PoS table. This could happen when extending the lexicon with entries that are homographs of other entries already available in the lexicon.

Only two properties are defined:

```
Property = "mapval" "="  INT

           | "iscombined" "=" "1" .
```

- `mapval`: the property `mapval` is mandatory and defines the integer value (<=255) used internally. This value is unique and constant; if the value for any symbol in the table is changed, new lingware should be regenerated accordingly.
- `iscombined`: set to 1 if the symbol represents a complex PoS tag.

The following section lists all PoS tags and their meanings currently defined in Pico. To know which PoS tags are actually available for each language consult the corresponding PoS table.

Productive Tag Suffixes

The PoS tags in the PoS Tag Description further down are described without their productive tag suffixes. The suffixes listed here are productive only by convention and the Pico engine does not have any built-in knowledge about morphological features.

Number

_SG = singular
_PL = plural

Gender

_M = masculine
_F = feminine

Examples

N_M_PL = noun masculine plural
V_SG = verb singular

PoS Tags Description

ABBREV	abbreviation
ADJ	adjective
ADV	adverb
ART	article
AUXB	"be" auxiliary verb
AUXH	"have" auxiliary verb
CARD	cardinal number
CLITIC	clitic
CONJ_CO	coordinating conjunction
CONJ_INF	infinitive conjunction (en-GB, en-US)
CONJ_QUE	"que" conjunction (fr-FR)
CONJ_SUB	subordinating conjunction
DEM	demonstrative
DET	determiner
DET_PRE	invariable indefinite (de-DE)
INDEF_ART	indefinite article
INDEF	indefinite
N	noun
NEG	negation
N_ADJ	noun-adjective
N_GEN	noun-genitive
N_ING	noun-gerund
PERS_PRO_OBJ	object personal pronoun
PERS_PRO_SUBJ	subject personal pronoun
POSS	possessive
POSTP	postposition
PREF_V	verb prefix
PREP	preposition
PRO	pronoun
PRON_ADV	pronominal adverb
PRO_REL	relative pronoun
PRO_V	pronoun + verb (en-GB, en-US)
PTCL	particle
Q_ADJ	interrogative adjective

Q_DET	interrogative determiner
Q_PRO	interrogative pronoun
Q_PTCL	interrogative particle
REL_PRO	relative pronoun
V	verb
V_AUX	auxiliary verb
V_AUX_INF	auxiliary verb (infinitive form)
V_AUX_PART	auxiliary verb (past participle form)
V_GER	verb (gerund form)
V_INF	verb (infinitive form)
V_MODAL	modal verb
V_MODAL_INF	modal verb (infinitive form)
V_MODAL_PART	modal verb (past participle form)
V_PART	verb (past participle form)
V_PART_PRES	verb (present participle form)
V_PAST	verb (past form)[1]

4.4 Grapheme Table

The Grapheme Table is needed for the tokenizing of the input character sequence. Tokenization is packing a single character or a group of consecutive characters into tokens of a specific type, based on the character properties defined in the grapheme properties file.

For each character a token type and an optional token id are defined. Following token types are available:

- WhiteSpaceCharacter
- VowelLikeCharacter
- ConsonantLikeCharacter
- DigitCharacter
- SequenceCharacter
- SingleCharacter

The tokenizer groups together sequences of characters of the same type with the following exceptions:

- VowelLikeCharacter and ConsonantLikeCharacter are also kept together
- SingleCharacter are always separated in one token per character
- if token ids are defined the tokenizer separates also characters of the same token type but with different token ids

The grapheme table syntax adheres to the general Property Table syntax (section 4.1):

```
GraphemesPropertyTableFile = { Entry } .

Entry =  ":SYM" Symbol  ":PROP" Property { "," Property } .
```

Properties are defined after the symbol and introduced by the keyword `:PROP`:

```
Property =
```

[1] **en-GB**: used only for lexical disambiguation in case of <read> [ri:d] vs. [red]. **en-US**: can be used freely to tag verbs in the past form (including past participle).


```
    "stoken" "=" STokenType
  | "stokenid" "=" STokenId
  | "punct" "=" PunctType
  | "graphsubs1" "=" Grapheme
  | "graphsubs2" "=" Grapheme
  .

Grapheme =
    '"' Single-UTF8-Character '"'
  | "'" Single-UTF8-Character "'"
.

STokenType =
    WhiteSpaceCharacter
  | VowelLikeCharacter
  | ConsonantLikeCharacter
  | DigitCharacter
  | SequenceCharacter
  | SingleCharacter
.

STokenId = INTEGER .

WhiteSpaceCharacter = "0" .
VowelLikeCharacter = "1" .
ConsonantLikeCharacter = "2" .
DigitCharacter = "3" .
SequenceCharacter = "4" .
SingleCharacter = "5" .
PunctType =
    SentenceFinalPunctuationMark
  | NotSentenceFinalPunctuationMark
.
SentenceFinalPunctuationMark = "2" .
NonSentenceFinalPunctuationMark  = "1" .
```


Example:

```
:SYM ","  :PROP stoken = 5, punct = 1
:SYM "-"  :PROP stoken = 4
:SYM "."  :PROP stoken = 5, punct = 2
:SYM "0"  :PROP stoken = 3
:SYM "1"  :PROP stoken = 3
:SYM "?"  :PROP stoken = 5, punct = 2
:SYM "@"  :PROP stoken = 4
:SYM "A"  :PROP stoken = 1, graphsubs1 = "a"
:SYM "B"  :PROP stoken = 2, graphsubs1 = "b"
```

5 Description of Text-Preprocessing Network Source Files

Applies to 1.2.1

The tokenized character sequence is fed into the preprocessing unit, which matches the input tokens with the tokens contained in the preprocessing network. All possible paths of matching input tokens with the tokens in the network are tried until no more paths are available. The best path found so far is then used to transform the input to the desired output.

A preprocessing network file consists of contexts, productions, tokens, attributes and their values, output items and strings.

The contexts define groups of active productions. Which context is active can be selected with a markup command in the input string. A context may consist of several entries in the context section of the network description. Each context entry defines the context name, the name of a network allowing to enable productions from other networks and the name of a production.

A production consists of a network of tokens describing a certain behavior, e.g. parsing a number with some constraints and giving it out directly or changing the input sequence to a desired output sequence. Productions are hierarchical as they can use other productions.

Tokens define which input sequence is accepted and what is given out instead of the input. They define also their successors and their alternative tokens. The behavior of a token (whether and what and how it is matching with an input token) is defined with attributes.

There are two kinds of attributes: those having values and those having no values. Depending on the attribute, the types of the attribute values can be integers, offsets to strings, offsets to production names, offsets to output items or offsets to lexical categories.

Output items define what is given out for a matched sequence of input tokens. Output items can define also the usage of built in functions like concatenation of strings or content of input tokens.

5.1 Preprocessing Network Description

A preprocessing network file starts with the keyword "NETWORK" followed by an offset to the string section defining the name of the network. Afterwards the sections for strings, lexical categories, attribute values, tokens, output items, productions and finally contexts are following.

Syntax:

```
PreprocessingNetworkFile =
  "NETWORK" NetNameOfs
  Strings
  LexCats
  AttrVals
  Tokens
  OutItems
```



```
  Productions
  Contexts
.
```

Remarks:

Preprocessing source files may contain the same kind of comments (line comments and range comments) as Property Tables source file, see (section 4.1).

5.2 STRINGS section

The STRINGS section contains all strings used in the network. That can be names of contexts, names of networks, names of productions, strings used for input token matching and strings used in the output items. The string section starts with the keyword "STRINGS" followed by the number of charcters (including the termination zeroes). Strings are always referenced with their offsets in the STRINGS section and are always zero terminated.

Syntax:

```
Strings =
  "STRINGS" Len
  StringsOfs String
  { StringsOfs String }
  "."
.
Len = UINT .
```

Remark:

The first entry starting at offset 0 has to be the empty string.

Example:

see CONTEXTS section.

5.3 CONTEXTS section

The context section contains as many entries as given with the number 'NrContextEntries', however the actual number of different contexts is given by the number of context entries with different context name offsets. A certain context consists of 4 numbers. The first number is an entry id, the second an offset to the context name, the third an offset to a network name and the fourth an offset to a production name. A context may consist of as many context entries as different productions should be enabled for this context. Productions from other networks may be enabled by using a different network name offset.

Syntax:

```
Contexts =
```



```
  "CONTEXTS" NrContextsEntries
  ContextEntry
  { ContextEntry }
  "."
.
NrContextEntries = UINT .
ContextEntry =
  ContextsOfs CtxNameOfs NetNameOfs ProdNameOfs
.
```

Remark:

The first context entry at offset 0 should contain only zeroes.

Example:

```
STRINGS 59
0 ""
1 "NETWORK1"
10 "CONTEXT1"
19 "CONTEXT2"
28 "CONTEXT3"
36 "PRODUCTION1"
48 "PRODUCTION2"
.
CONTEXTS 5
0 0 0 0
1 10 1 36   ! CONTEXT1 using PRODUCTION1 of network NETWORK1
2 19 1 48   ! CONTEXT2 using PRODUCTION2 of network NETWORK1
3 28 1 36   ! CONTEXT3 using PRODUCTION1 of network NETWORK1 and
4 28 1 48   !                PRODUCTION2 of network NETWORK1
.
```

If a context should be empty, an entry referencing the empty string for network and production name can be used.

5.4 PRODUCTIONS section

The production entries in the production section consist of 5 numbers. The first number 'ProductionOfs' is the entry index or offset in the production section. The second number

'ProdPrefCost' is a cost value to prefer or penalize the selection of the production against other productions. The third number 'ProdNameOfs' is the offset to the production name. The fourth and fifth number 'ATokOfs' and 'ETokOfs' are offsets to the start and ending tokens of the production.

Syntax:

```
Productions =
  "PRODUCTIONS" NrProductions
  Production
  { Production }
  "."
.

Production =
  ProductionsOfs ProdPrefCost ProdNameOfs ATokOfs ETokOfs
.

NrProductions = UINT .
ProdPrefCost = INT .
ATokOfs = TokensOfs .
ETokOfs = TokensOfs .
```

Remark:

The first production entry at offset 0 should contain only zeroes.

Example:

```
STRINGS 59
0 ""
1 "NETWORK1"
10 "CONTEXT1"
19 "CONTEXT2"
28 "CONTEXT3"
36 "PRODUCTION1"
48 "PRODUCTION2"
.

TOKENS ...
```



```
0 {} 0 0 0 0
1 ...
2 ...
3 ...
4 ...
...
.

PRODUCTIONS 2
0 0 0 0 0
1 0 36 1 2  ! PRODUCTION2 starting at token 1 and ending at token 2
2 0 48 3 4  ! PRODUCTION2 starting at token 3 and ending at token 4
.

CONTEXTS 5
0 0 0 0
1 10 1 36   ! CONTEXT1 using PRODUCTION1 of network NETWORK1
2 19 1 48   ! CONTEXT2 using PRODUCTION2 of network NETWORK1
3 28 1 36   ! CONTEXT3 using PRODUCTION1 of network NETWORK1 and
4 28 1 48   !                PRODUCTION2 of network NETWORK1
.
```

5.5 TOKENS section

The TOKENS section contains 'NrTokens' tokens. Each token entry consists of a set of attributes, an offset to the first attribute value ('AttrValsOfs') and three fields ('NextTokenOfs', 'AltLTokenOfs', 'AltRTokenOfs') for connecting the token with other tokens to be able to form a network. With the 'NextTokenOfs' field the tokens can be sequenced. With the 'AltLTokenOfs' and the 'AltRTokenOfs' field tokens can form an alternative.

Syntax:

```
Tokens =
  "TOKENS" NrTokens
  Token { Token }
  "."
.
```



```
  Token =
    TokensOfs AttributeSet NextTokenOfs AltLTokenOfs AltRTokenOfs
AttrValsOfs
  .

  NrTokens = UINT .
  NextTokenOfs = TokensOfs .
  AltLTokenOfs = TokensOfs .
  AltRTokenOfs = TokensOfs .

  Attribute =
      "TSEOut"     | "TSEMin"   | "TSEMax"   | "TSELen"    | "TSEVal"
    | "TSEStr"     | "TSEHead"  | "TSEMid"   | "TSETail"   | "TSEProd"
    | "TSEProdExt" | "TSEVar"   | "TSELex"   | "TSECost"   | "TSEID"
    | "TSEBegin"   | "TSEEnd"   | "TSESpace" | "TSEDigit"  | "TSELetter"
    | "TSEChar"    | "TSESeq"   | "TSECmpr"  | "TSENLZ"    | "TSERoman"
    | "TSECI"      | "TSECIS"   | "TSEAUC"   | "TSEALC"    | "TSESUC"
    | "TSEAccept"  | "TSENext"  | "TSEAltL"  | "TSEAltR"
  .

  AttributeSet = "{" Attribute { "," Attribute } "}" .
```

Remark:

The first attribute entry at offset 0 should be an empty entry, eg

```
0 {} 0 0 0 0
```

The attributes define among other things

- how to match a token with an input token or
- what to give out or
- how to connect the token to other tokens or
- how to influence the rating of certain path

Attributes without values are:

TSEBegin	matches with first token since a channel has been opened
TSEEnd	matches with the last token before a channel is closed
TSESpace	matches tokens of type W (whitespace (stoken type 0))

TSELetter	matches tokens of type L (letter (stoken type 1 or 2))
TSEDigit	matches tokens of type D (digit (stoken type 3))
TSESeq	matches tokens of type S (character sequence(stoken type 4))
TSEChar	matches tokens of type C (single character (stoken type 5))
TSECmpr	has always to be set
TSENLZ	matches D tokens without leading zeroes
TSERoman	matches L tokens containing roman numbers
TSECI	matches L tokens case insensitive
TSECIS	matches L tokens case insensitive at start (only first character)
TSEAUC	matches L tokens with only upper case characters
TSEALC	matches L tokens with only lower case characters
TSESUC	matches L tokens with an upper case character as first character
TSEAccept	accepting token marking the matched end of a production
TSENext	token has a follower token specified in the NextTokenOfs field
TSEAltL	token has a alternative token specified in the AltLTokenOfs field which is alphabetically smaller than the actual input token.
TSEAltR	token has a alternative token specified in the AltRTokenOfs field which is alphabetically larger than the actual input token.

Attributes with values are:

TSEMin	matches tokens of type W, D, S and L having the minimal length specified in the attribute value (INT)
TSEMax	matches tokens of type W, D, S and L having a maximal length specified in the attribute value (INT)
TSELen	matches tokens of type W, D, S and L having the exact length specified in the attribute value (INT)
TSEVal	matches D tokens with same values as specified in the attribute value (INT)
TSEStr	matches W, D, S, L and C tokens with same character sequence as specified in the string the attribute value points to (StringOfs)
TSEHead	matches D, S and L tokens starting with the same character sequence as specified in the string the attribute value points to (StringOfs)
TSEMid	matches D, S and L tokens containing the character sequence as specified in the string the attribute value points to (StringOfs)
TSETail	matches D, S and L tokens ending with the same character sequence as specified in the string the attribute value points to (StringOfs)
TSEProd	matches the following input tokens with the production the attribute value

	points to (ProductionsOfs)
TSEProdExt	matches the following input tokens with an external production from another network having the network and production name offset defined in attribute value (StringOfs)
TSELex	matches L tokens having an entry in the lexicon with the lexical categories pointed to by the attribute value (LexCatsOfs)
TSEOut	make an output according to what is defined in the output item specified in the attribute value (OutItemsOfs)
TSEVar	variable with the id specified in the attribute value containing the input token or the result of a matched production (INT)
TSECost	associating a cost value to the token (INT)
TSEID	matches tokens with the same stokenid as specified in the attribute value (INT)

The attribute values for a certain set of attributes are starting at position position 'AttrValsOfs'. Given the above order of the attributes, the i-th attribute (i=1..n) uses the attribute value at 'AttrValOfs'+i-1.

5.6 ATTRVALS section

An attribute value is an integer or an offset to another data section depending on the type of the attribute. The attribute values for a certain attribute set are stored consecutively.

Syntax:

```
AttrVals =
    "ATTRVALS" NrAttrVals
    AttrValsOfs AttrVal
    { AttrValsOfs AttrVal }
    "."
  .

AttrVal = INT | StringsOfs | ProductionsOfs | OutItemsOfs | LexCatsOfs .
```

Remark:

The first attribute value entry at offset 0 has to be a 0.

5.7 OUTITEMS section

The out item section defines what is given out from a token with a TSEOut attribute. Therefore the attribute value of the TSEOut attribute points to an out item entry in the out

items section. Out items can be nested so the output may be the input argument for other output items.

What is returned or given out by an out item is defined by the out item type. VAL or STR just return the integer value or the string offset. Other out items define some functions like LEFT, RIGHT or CONCAT on the input argument(s). Some of these functions do not produce direct output but influence the way a certain text is synthesized by emitting a command. Such functions more or less correspond to the markups available on textual level, like it is the case with SPEED, PITCH and VOLUME. Some of these 'markup-like' functions are reserved for future versions of PICO but are ignored right now.

Out items consist of three fields:

- OutItemType: type of out item
- Arg: argument value depending on out item type
- NextArg: offset pointing to the next argument

Following out item types are available:

STR	emit string pointed to by Arg
VAL	emit integer value in Arg
VAR	emit context of variable Arg
CONCAT	emit concatenation of x,y,... Concatenate list of strings with first element pointed to by argument Arg and the next by NextArg of Arg
ITEM	emit x-th element of argument y; x in argument Arg pointed to by Arg. y in argument Arg pointed to by NextArg of Arg pointed to by Arg.
RIGHT	emit right n characters of string x; in argument Arg pointed to by Arg. x in argument Arg pointed to by NextArg of Arg pointed to by Arg.
LEFT	emit left n characters of string x; n in argument Arg pointed to by Arg. x in argument Arg pointed to by NextArg of Arg pointed to by Arg.
SPELL	emit spelled sequence of string arg1 with pauses in arg2
SVOXPA	not available (ignored) in PICO emit phoneme command with alphabet SVOXPA and phone in argument Arg pointed to by Arg.
SAMPA	not available (ignored) in PICO emit phoneme command with alphabet SAMPA and phone sequence in argument Arg pointed to by Arg.
AUDIOEDIT	not available in PICO
RLZ	remove leading zeros of argument Arg pointed to by Arg and emit result
IGNORE	emit ignore command (Arg=0: ignore start, Arg=1: ignore end)
SENTENCE	Arg=0: emit sentence start command Arg =1: emit sentence end command
PARAGRAPH	Arg=0: emit paragraph end command Arg =1: emit paragraph start command
ROMAN	emit string with cardinal aquivalent of roman number in argument Arg pointed to by Arg.

GENFILE	Arg>0: emit genfile command with file in argument Arg to by Arg Arg=0: emit genfile end command
PLAY	Arg>0: emit play command with file in argument Arg pointed to by Arg Arg=0: emit play end command
USESIG	Arg>0: emit usesig start command with file in argument Arg pointed to by Arg Arg=0: emit usesig end command
SPEED	emit set speed command to level to level in argument Arg pointed to by Arg.
PITCH	emit set pitch command to level in argument Arg pointed to by Arg.
VOLUME	emit set volume command to level in argument Arg pointed to by Arg.
VOICE	not available (ignored) in PICO emit set voice command to voice name in argument Arg pointed to by Arg.
MARK	not available (ignored) in PICO emit mark command with string in argument Arg pointed to by Arg.
BREAK	emit break command with duration in argument Arg pointed by Arg (in ms).
CONTEXT	emit set context command to context in argument Arg pointed by Arg (in ms).

Syntax:

```
OutItems =
  "OUTITEMS" NrOutItems
  OutItem { OutItem }
  "."
.

OutItem =
  OutItemOfs NextArg OutItemType Arg
.

NextArg = OutItemsOfs .
NrOutItems = UINT .

OutItemType =
      "AUDIOEDIT"   | "BREAK"       | "CONCAT"      | "CONTEXT"
    | "GENFILE"     | "IGNORE"      | "ITEM"        | "LEFT"
```



```
    | "MARK"        | "PARA"        | "SAMPA"       | "SVOXPA"
    | "PITCH"       | "PLAY"        | "RLZ"         | "RIGHT"
    | "ROMAN"       | "SENT"        | "SPEED"       | "SPELL"
    | "STR"         | "USESIG"      | "VAL"         | "VAR"
    | "VOICE"       | "VOLUME"
.

Arg =
      INT
    | StringOfs
    | OutItemOfs
.
```

Argument description of out items:

5.8 LEXCATS section

The LEXCATS section defines the id's of lexical categories. As lexical categories are not yet supported in PICO, this section contains normally only one entry with value 0.

Syntax:

```
LexCats =
  "LEXCATS" NrLexCats
  LexCatEntry
  { LexCatEntry }
  "."
.

LexCatEntry = LexCatsOfs LexCat { LexCat } .
LexCat = INT16 .
```

Remarks:

The first LexCatEntry at offset 0 should have only one LexCat with value 0.

The last lexical category of a LexCatEntry should be a 0.

6 Description of Lexicon Source File

Applies to 1.2.5

6.1 Introduction

The Pico lexicon a list of entries that are considered to be exceptions and corrections to the general mechanism of PoS and G2P prediction built into Pico. The main applications for the lexicon are:

- to support the preprocessing, especially in the number reading domain
- to read characters or character sequences that were not included in the G2P training
- to correct errors in the part of speech (PoS) prediction
- and to correct errors in the G2P prediction.

6.2 Format

The lexicon is a list of entries, with one entry per line. An entry consists of three fields, separated by white space:

```
Lexicon = { Entry } .

Entry = PoS Ortho ( Phono | ":G2P" ) .
```

The first field `PoS` describes the Part-of-Speech of the entry. The PoS might be also composed (ambiguous, e.g. `N^V`). Please refer to the corresponding PoS table to find out which PoS are supported for a given language.

The second field `Ortho` represents the orthographic form of the word and is enclosed in double quotes.

The third field represents the phonetic transcription for the given combination of Part-of-.Speech and /orthographic . Two variants are supported: the full phonetic representation `Phono` enclosed in double quotes, and the `:G2P` keyword that leaves the phonetic transcription of the word to be predicted by the G2P prediction tree. This second variant should be used when only the PoS for a given word is predicted wrongly and is useful to keep the size of the lexicon small.

6.3 Restrictions

The following restrictions apply:

1. Only up to 5 homographs (five entries having the same orthographic string) are supported.
2. Homographs, i.e. an orthographic form with several different phonetic transcriptions, may not be described using combined PoS tags or using the G2P keyword.

Example:

The following entries for the orthographic word “second” will lead to unexpected results:

```
ADJ^ADV^N "second" :G2P
```



```
V "second" "sIk'And"
```

Instead, the composed tag should be split, resulting in separate entries in the lexicon, and the `G2P` keyword has to be replaced by explicit phonetic transcriptions:

```
ADJ "second" "s'ek@nd"
ADV "second" "s'ek@nd"
N "second" "s'ek@nd"
V "second" "sIk'And"
```

7 Description of PDF Source Files

Applies to 1.2.18 ... 1.2.21

7.1 Introduction

Part of the signal generation lingware enables Pico to determine acoustic speech parameters including:

- Phone overall duration and HMM phone model individual state durations
- HMM phone model individual state F0 values
- HMM phone model individual state mel cepstral parameter vectors
- HMM phone model individual state excitation parameters
- HMM phone model individual state phase parameters.

This part of the lingware is implemented with 4 different resources:

- Duration resource (DUR)
- Pitch resource (LFZ)
- Cepstral resource (MGC, includes excitation)
- Phase resource (PHA)

7.2 Internal Format of the pkb

The internal format of the binary resources is described on the corresponding textual input representation. The identifier, number of items and data type is specified for each resource component in the correct order. Comments are added at the end of the lines, if suitable, to better explain the content. In this document we add some information that is not easy to insert in the textual description comments.

7.3 Duration Resource

This resource is used on the system to retrieve phone and HMM phone individual state durations starting from the indices predicted by the corresponding DT traversal. Durations in this resource are specified in number of frames, where the frame length is 4 ms. The decision tree is used to determine the duration index per phoneme, and the duration index points to a vector containing the total duration of the phoneme and the duration of each phoneme state. The structure of this pdf is reported in the following:

numvectors	total number of vectors in the pdf file
vecsize	length of each vector
sampperframe	number of samples per frame
phonquantlen	length of the phone quantization table
phonquant	phone quantization table
statequantlen	length of the state quantization table

statequant state quantization table

For more details as the number of entries, the data type of each entry, and numeric examples, see the corresponding textual representation source.

7.4 Pitch Resource

This resource is used on the system to retrieve phone level and HMM phone individual state pitch values starting from the indices predicted by the corresponding DT traversal. F0 values in this resource are specified in Hertz. The decision tree is used to determine the F0 indices per each phoneme state. There is, in fact, a separate decision tree for each state. The F0 index points then to a vector in the LFZ pdf table. Each LFZ vector contains a static log F0 value, delta log F0 value, and delta delta log F0 value, and corresponding inverse variance values. The delta and variance information are used to smooth the predicted F0 values based on a maximum likelihood criterion.

The structure of this pdf is reported in the following:

numvectors total number of vectors in the pdf file

vecsize length of each vector

numstates number of states

numframesperstate number of vectors per state

ceporder length of static part of vector

numvuv length of voicing part of vector

numdelta length of delta part of vector

meanpow log fixed-point base for stored vectors

maxbigpow maximum log fixed-point base for smoothing

amplif correction factor in percent for synthesized speech or smoothed f0

meanpowum log scaling factor for each stored cepstral dimension

ivarpow log scaling factor for inverse variances

content psf content

For more details as the number of entries, the data type of each entry, and numeric examples, see the corresponding textual representation source.

7.5 Cepstral and Phase Resources

A third set of decision trees predicts a spectral index for each state. The spectral index points to a vector in the MGC pdf table and to a vector in the phase pdf table. Each MGC vector contains a voicing byte, static mel cepstral values, delta and delta delta cepstral values, and corresponding inverse variance values. The delta and variance information are used to smooth the predicted MGC values based on a maximum likelihood criterion. The spectral index also points to a phase vector, which is used for synthesis of a natural voice quality.

The structure of MGC pdf is reported in the following:

numvectors	total number of vectors in the pdf file
vecsize	length of each vector
numstates	number of states
numframesperstate	number of vectors per state
ceporder	length of static part of vector
numvuv	length of voicing part of vector
numdelta	length of delta part of vector
meanpow	log fixed point base for stored vectors
maxbigpow	maximum log fixed point base for smoothing
amplif	correction factor in percent for synthesized speech or smoothed f0
meanpowum	log scaling factor for each stored cepstral dimension
ivarpow	log scaling factor for inverse variances
content	cepstral content

The structure of Phase pdf is reported in the following:

numvectors total number of vectors in the pdf file

index phase indices table

n_comps	phase sub table n.0 number of components
phase	phase sub table n.0
...	
...	
n_comps	phase sub table n.numvectors-1 number of components
phase	phase sub table n.numvectors-1

For more details as the number of entries, the data type of each entry, and numeric examples, see the corresponding textual representation source.

7.6 Data Format for PDF Resource Values

In the pdf files textual description, data values are expected to be UNSIGNED, 8-, 16- or 32-bit values. In general, this is the correct type of data for ‘offsets’ or for most of the data values.

If some of the data is related to a signed quantity, please note that:

- Unsigned char

  unsigned values 0..127 maps to 0..127 positive values;

  unsigned values 255..128 maps to negative values (-1..-128)

- Unsigned short

  unsigned values 0..32767 maps to 0..32767 positive values;

unsigned values 65536..32768 maps to negative values (-1..-32768)

- Unsigned int

unsigned values 0..2**31-1 maps to 0..2**31-1 positive values;

unsigned values 2**32..2**31 maps to negative values (-1..2**31)

Appendix

A. Syntax of Preprocessing Network File (in UTF-8 format)

```
PreprocessingNetworkFile =
  "NETWORK" NetNameOfs
  Strings
  LexCats
  AttrVals
  Tokens
  OutItems
  Productions
  Contexts
.

##############################################################################

String = UTF8STRING .
StringsOfs = UINT .
LexCatsOfs = UINT .
AttrValsOfs = UINT .
OutItemsOfs = UINT .
TokensOfs = UINT .
ProductionsOfs = UINT .
ContextsOfs = UINT .

CtxNameOfs = StringsOfs .
NetNameOfs = StringsOfs .
ProdNameOfs = StringsOfs .
Len = UINT .

##############################################################################

Strings =
```



```
  "STRINGS" Len
  StringsOfs String
  { StringsOfs String }
  "."
.

##############################################################################

NrLexCats = UINT .
LexCat = INT16 .

LexCats =
  "LEXCATS" NrLexCats
  LexCatsOfs { LexCat } "0"
  { LexCatsOfs { LexCat } "0" }
  "."
.

##############################################################################

AttrVal = INT | StringsOfs | ProductionsOfs | OutItemsOfs | LexCatsOfs .
NrAttrVals = UINT .

AttrVals =
  "ATTRVALS" NrAttrVals
  AttrValsOfs AttrVal
  { AttrValsOfs AttrVal }
  "."
.

##############################################################################

NrOutItems = UINT .
NextArg = OutItemsOfs .
```



```
OutItemType =
      "AUDIOEDIT"   | "BREAK"       | "CONCAT"      | "CONTEXT"
    | "GENFILE"     | "IGNORE"      | "ITEM"        | "LEFT"
    | "MARK"        | "PARA"        | "SAMPA"       | "SVOXPA"
    | "PITCH"       | "PLAY"        | "RLZ"         | "RIGHT"
    | "ROMAN"       | "SENT"        | "SPEED"       | "SPELL"
    | "STR"         | "USESIG"      | "VAL"         | "VAR"
    | "VOICE"       | "VOLUME"
.

Arg =
      INT
    | StringOfs
    | OutItemOfs
.

OutItem =
  OutItemOfs NextArg OutItemType Arg
.

OutItems =
  "OUTITEMS" NrOutItems
  OutItem { OutItem }
  "."
.

##############################################################################

NrTokens = UINT .
NextTokenOfs = TokensOfs .
AltLTokenOfs = TokensOfs .
AltRTokenOfs = TokensOfs .

Attribute =
    "TSEOut"     | "TSEMin"   | "TSEMax"   | "TSELen"    | "TSEVal"
```



```
  | "TSEStr"     | "TSEHead"  | "TSEMid"   | "TSETail"   | "TSEProd"
  | "TSEProdExt" | "TSEVar"   | "TSELex"   | "TSECost"   | "TSEID"
  | "TSEBegin"   | "TSEEnd"   | "TSESpace" | "TSEDigit"  | "TSELetter"
  | "TSEChar"    | "TSESeq"   | "TSECmpr"  | "TSENLZ"    | "TSERoman"
  | "TSECI"      | "TSECIS"   | "TSEAUC"   | "TSEALC"    | "TSESUC"
  | "TSEAccept"  | "TSENext"  | "TSEAltL"  | "TSEAltR"
.

AttributeSet = "{" Attribute { "," Attribute } "}"



Token =
  TokensOfs AttributeSet NextTokenOfs AltLTokenOfs AltRTokenOfs
AttrValsOfs
.

Tokens =
  "TOKENS" NrTokens
  Token { Token }
  "."
.

##############################################################################

NrProductions = UINT .
ProdPrefCost = INT .
ATokOfs = TokensOfs .
ETokOfs = TokensOfs .

Production =
  ProductionsOfs ProdPrefCost ProdNameOfs ATokOfs ETokOfs
.

Productions =
  "PRODUCTIONS" NrProductions
```



```
  Production
  { Production }
  "."
.
##############################################################################

NrContextsEntries = UINT .

Context =
  ContextsOfs CtxNameOfs NetNameOfs ProdNameOfs
.

Contexts =
  "CONTEXTS" NrContextsEntries
  Context
  { Context }
  "."
.
```

