Go to the first, previous, next, last section, table of contents.

---

14 Utterances

The utterance structure lies at the heart of Festival. This chapter describes its basic form and the functions available to manipulate it.

14.1 Utterance structure

Festival's basic object for synthesis is the *utterance*. An represents some chunk of text that is to be rendered as speech. In general you may think of it as a sentence but in many cases it wont actually conform to the standard linguistic syntactic form of a sentence. In general the process of text to speech is to take an utterance which contaisn a simple string of characters and convert it step by step, filling out the utterance structure with more information until a waveform is built that says what the text contains.

The processes involved in convertion are, in general, as follows

Tokenization
: Converting the string of characters into a list of tokens. Typically this means whitespace separated tokesn of the original text string.

Token identification
: identification of general types for the tokens, usually this is trivial but requires some work to identify tokens of digits as years, dates, numbers etc.

Token to word
: Convert each tokens to zero or more words, expanding numbers, abbreviations etc.

Part of speech
: Identify the syntactic part of speech for the words.

Prosodic phrasing
: Chunk utterance into prosodic phrases.

Lexical lookup
: Find the pronucnation of each word from a lexicon/letter to sound rule system including phonetic and syllable structure.

Intonational accents
: Assign intonation accents to approrpiate syllables.

Assign duration
: Assign duration to each phone in the utterance.

Generate F0 contour (tune)
: Generate tune based on accents etc.

Render waveform
: Render waveform from phones, duration and F) target values, this itself may take several steps including unit selection (be they diphones or other sized units), imposition of dsesired prosody (duration and F0) and waveform reconstruction.

The number of steps and what actually happens may vary and is dependent on the particular voice selected and the utterance's *type*, see below.

Each of these steps in Festival is achived by a *module* which will typically add new information to the utterance structure.

An utterance structure consists of a set of *items* which may be part of one or more *relations*. Items represent things like words and phones, though may also be used to represent less concrete objects like noun phrases, and nodes in metrical trees. An item contains a set of features, (name and value). Relations are typically simple lists of items or trees of items. For example the the `Word` relation is a simple list of items each of which represent a word in the utternace. Those words will also be in other relations, such as the *SylStructure* relation where the word will be the top of a tree structure containing its syllables and segments.

Unlike previous versions of the system items (then called stream items) are not in any particular relations (or stream). And are merely part of the relations they are within. Importantly this allows much more general relations to be made over items that was allowed in the previous system. This new architecture is the continuation of our goal of providing a general efficient structure for representing complex interrelated utterance objects.

The architecture is fully general and new items and relations may be defined at run time, such that new modules may use any relations they wish. However within our standard English (and other voices) we have used a specific set of relations ass follows.

Token
: a list of trees. This is first formed as a list of tokens found in a character text string. Each root's daughters are the *Word*'s that the token is related to.

Word
: a list of words. These items will also appear as daughters (leaf nodes) of the `Token` relation. They may also appear in the `Syntax` relation (as leafs) if the parser is used. They will also be leafs of the `Phrase` relation.

Phrase
: a list of trees. This is a list of phrase roots whose daughters are the `Word's` within those phrases.

Syntax
: a single tree. This, if the probabilistic parser is called, is a syntactic binary branching tree over the members of the `Word` relation.

SylStructure
: a list of trees. This links the `Word`, `Syllable` and `Segment` relations. Each `Word` is the root of a tree whose immediate daughters are its syllables and their daughters in turn as its segments.

Syllable
: a list of syllables. Each member will also be in a the `SylStructure` relation. In that relation its parent will be the word it is in and its daughters will be the segments that are in it. Syllables are also in the `Intonation` relation giving links to their related intonation events.

Segment
: a list of segments (phones). Each member (except silences) will be leaf nodes in the `SylStructure` relation. These may also be in the `Target` relation linking them to F0 target points.

IntEvent
: a list of intonation events (accents and bounaries). These are related to syllables through the `Intonation` relation as leafs on that relation. Thus their parent in the `Intonation` relation is the syllable these events are attached to.

Intonation
: a list of trees relating syllables to intonation events. Roots of the trees in `Intonation` are `Syllables` and their daughters are `IntEvents`.

Wave
: a single item with a feature called `wave` whose value is the generated waveform.

This is a non-exhaustive list some modules may add other relations and not all utterance may have all these relations, but the above is the general case.

14.2 Utterance types

The primary purpose of types is to define which modules are to be applied to an utterance. `UttTypes` are defined in `` `lib/synthesis.scm' ``. The function `defUttType` defines which modules are to be applied to an utterance of that type. The function `utt.synth` is called applies this list of module to an utterance before waveform synthesis is called.

For example when a `Segment` type Utterance is synthesized it needs only have its values loaded into a `Segment` relation and a `Target` relation, then the low level waveform synthesis module `Wave_Synth` is called. This is defined as follows

```
(defUttType Segments
  (Initialize utt)
```

```
  (Wave_Synth utt))
```

A more complex type is `Text` type utterance which requires many more modules to be called before a waveform can be synthesized

```
(defUttType Text
  (Initialize utt)
  (Text utt)
  (Token utt)
  (POS utt)
  (Phrasify utt)
  (Word utt)
  (Intonation utt)
  (Duration utt)
  (Int_Targets utt)
  (Wave_Synth utt)
)
```

The `Initialize` module should normally be called for all types. It loads the necessary relations from the input form and deletes all other relations (if any exist) ready for synthesis.

Modules may be directly defined as C/C++ functions and declared with a Lisp name or simple functions in Lisp that check some global parameter before calling a specific module (e.g. choosing between different intonation modules).

These types are used when calling the function `utt.synth` and individual modules may be called explicitly by hand if required.

Because we expect waveform synthesis methods to themselves become complex with a defined set of functions to select, join, and modify units we now support an addition notion of `SynthTypes` like `UttTypes` these define a set of functions to apply to an utterance. These may be defined using the `defSynthType` function. For example

```
(defSynthType Festival
  (print "synth method Festival")

  (print "select")
  (simple_diphone_select utt)

  (print "join")
  (cut_unit_join utt)

  (print "impose")
  (simple_impose utt)
  (simple_power utt)

  (print "synthesis")
  (frames_lpc_synthesis utt)
  )
```

A `SynthType` is selected by naming as the value of the parameter `Synth_Method`.

Duration the application of the function `utt.synth` there are three hooks applied. This allows addition control of the synthesis process. `before_synth_hooks` is applied before any modules are applied. `after_analysis_hooks` is applied at the start of `Wave_Synth` when all text, linguistic and prosodic processing have been done. `after_synth_hooks` is applied after all modules have been applied. These are useful for things such as, altering the volume of a voice that happens to be quieter than others, or for example outputing information for a talking head before waveform synthesis occurs so preparation of the facial frames and synthesizing the waveform may be done in parallel. (see `` `festival/examples/th-mode.scm' `` for an example use of these hooks for a talking head text mode.)

14.3 Example utterance types

A number of utterance types are currently supported. It is easy to add new ones but the standard distribution includes the following.

`Text`
Raw text as a string.

```
(Utterance Text "This is an example")
```

`Words`
A list of words

```
(Utterance Words (this is an example))
```

Words may be atomic or lists if further features need to be specified. For example to specify a word and its part of speech you can use

```
(Utterance Words (I (live (pos v)) in (Reading (pos n) (tone H-H%))))
```

Note: the use of the tone feature requires an intonation mode that supports it. Any feature and value named in the input will be added to the Word item.

`Phrase`
This allows explicit phrasing and features on Tokens to be specified. The input consists of a list of phrases each contains a list of tokens.

```
(Utterance
 Phrase
 ((Phrase ((name B))
   I saw the man
   (in ((EMPH 1)))
   the park)
  (Phrase ((name BB))
   with the telescope)))
```

ToBI tones and accents may also be specified on Tokens but these will only take effect if the selected intonation method uses them.

`Segments`
This allows specification of segments, durations and F0 target values.

```
(Utterance
 Segments
 ((# 0.19 )
  (h 0.055 (0 115))
  (@ 0.037 (0.018 136))
  (l 0.064 )
  (ou 0.208 (0.0 134) (0.100 135) (0.208 123))
  (# 0.19)))
```

Note the times are in *seconds* NOT milliseconds. The format of each segment entry is segment name, duration in seconds, and list of target values. Each target value consists of a pair of point into the segment (in seconds) and F0 value in Hz.

`Phones`
This allows a simple specification of a list of phones. Synthesis specifies fixed durations (specified in `FP_duration`, default 100 ms) and monotone intonation (specified in `FP_F0`, default 120Hz). This may be used for simple checks for waveform synthesizers etc.

```
(Utterance Phones (# h @ l ou #))
```

Note the function `SayPhones` allows synthesis and playing of lists of phones through this utterance type.

`Wave`
A waveform file. Synthesis here simply involves loading the file.

```
(Utterance Wave fred.wav)
```

Others are supported, as defined in `` `lib/synthesis.scm' `` but are used internally by various parts of the system. These include `Tokens` used in TTS and `SegF0` used by `utt.resynth`.

14.4 Utterance modules

The module is the basic unit that does the work of synthesis. Within Festival there are duration modules, intonation modules, wave synthesis modules etc. As stated above the utterance type defines the set of modules which are to be applied to the utterance. These modules in turn will create relations and items so that ultimately a waveform is generated, if required.

Many of the chapters in this manual are solely concerned with particular modules in the system. Note that many modules have internal choices, such as which duration method to use or which intonation method to use. Such general choices are often done through the `Parameter` system. Parameters may be set for different features like `Duration_Method`, `Synth_Method` etc. Formerly the values for these parameters were atomic values but now they may be the functions themselves. For example, to select the Klatt duration rules

```
(Parameter.set 'Duration_Method Duration_Klatt)
```

This allows new modules to be added without requiring changes to the central Lisp functions such as `Duration`, `Intonation`, and `Wave_Synth`.

14.5 Accessing an utterance

There are a number of standard functions that allow one to access parts of an utterance and traverse through it.

Functions exist in Lisp (and of course C++) for accessing an utterance. The Lisp access functions are

`(utt.relationnames UTT)'
: returns a list of the names of the relations currently created in `UTT`.

`(utt.relation.items UTT RELATIONNAME)'
: returns a list of all items in `RELATIONNAME` in `UTT`. This is nil if no relation of that name exists. Note for tree relation will give the items in pre-order.

`(utt.relation_tree UTT RELATIONNAME)'
: A Lisp tree presentation of the items `RELATIONNAME` in `UTT`. The Lisp bracketing reflects the tree structure in the relation.

`(utt.relation.leafs UTT RELATIONNAME)'
: A list of all the leafs of the items in `RELATIONNAME` in `UTT`. Leafs are defined as those items with no daughters within that relation. For simple list relations `utt.relation.leafs` and `utt.relation.items` will return the same thing.

`(utt.relation.first UTT RELATIONNAME)'
: returns the first item in `RELATIONNAME`. Returns `nil` if this relation contains no items

`(utt.relation.last UTT RELATIONNAME)'
: returns the last (the most next) item in `RELATIONNAME`. Returns `nil` if this relation contains no items

`(item.feat ITEM FEATNAME)'
: returns the value of feature `FEATNAME` in `ITEM`. `FEATNAME` may be a feature name, feature function name, or pathname (see below). allowing reference to other parts of the utterance this item is in.

`(item.features ITEM)'
: Returns an assoc list of feature-value pairs of all local features on this item.

`(item.name ITEM)'
: Returns the name of this `ITEM`. This could also be accessed as `(item.feat ITEM 'name)`.

`(item.set_name ITEM NEWNAME)'
: Sets name on `ITEM` to be `NEWNAME`. This is equivalent to `(item.set_feat ITEM 'name NEWNAME)`

`(item.set_feat ITEM FEATNAME FEATVALUE)'
: set the value of `FEATNAME` to `FEATVALUE` in `ITEM`. `FEATNAME` should be a simple name and not refer to next, previous or other relations via links.

`(item.relation ITEM RELATIONNAME)'
: Return the item as viewed from `RELATIONNAME`, or `nil` if `ITEM` is not in that relation.

`(item.relationnames ITEM)'
: Return a list of relation names that this item is in.

`(item.relationname ITEM)'
: Return the relation name that this item is currently being viewed as.

`(item.next ITEM)'

Return the next item in ITEM's current relation, or nil if there is no next.

`(item.prev ITEM)'
Return the previous item in ITEM's current relation, or nil if there is no previous.

`(item.parent ITEM)'
Return the parent of ITEM in ITEM's current relation, or nil if there is no parent.

`(item.daughter1 ITEM)'
Return the first daughter of ITEM in ITEM's current relation, or nil if there are no daughters.

`(item.daughter2 ITEM)'
Return the second daughter of ITEM in ITEM's current relation, or nil if there is no second daughter.

`(item.daughtern ITEM)'
Return the last daughter of ITEM in ITEM's current relation, or nil if there are no daughters.

`(item.leafs ITEM)'
Return a list of all lefs items (those with no daughters) dominated by this item.

`(item.next_leaf ITEM)'
Find the next item in this relation that has no daughters. Note this may traverse up the tree from this point to search for such an item.

As from 1.2 the utterance structure may be fully manipulated from Scheme. Relations and items may be created and deleted, as easily as they can in C++;

`(utt.relation.present UTT RELATIONNAME)'
returns t if relation named RELATIONNAME is present, nil otherwise.

`(utt.relation.create UTT RELATIONNAME)'
Creates a new relation called RELATIONNAME. If this relation already exists it is deleted first and items in the relation are derefenced from it (deleting the items if they are no longer referenced by any relation). Thus create relation guarantees an empty relation.

`(utt.relation.delete UTT RELATIONNAME)'
Deletes the relation called RELATIONNAME in utt. All items in that relation are derefenced from the relation and if they are no longer in any relation the items themselves are deleted.

`(utt.relation.append UTT RELATIONNAME ITEM)'
Append ITEM to end of relation named RELATIONNAME in UTT. Returns nil if there is not relation named RELATIONNAME in UTT otherwise returns the item appended. This new item becomes the last in the top list. ITEM item may be an item itself (in this or another relation) or a LISP description of an item, which consist of a list containing a name and a set of feature vale pairs. It ITEM is nil or inspecified an new empty item is added. If ITEM is already in this relation it is dereferenced from its current position (and an emtpy item re-inserted).

`(item.insert ITEM1 ITEM2 DIRECTION)'
Insert ITEM2 into ITEM1's relation in the direction specified by DIRECTION. DIRECTION may take the value, before, after, above and below. If unspecified, after is assumed. Note it is not recommended to insert above and below and the functions item.insert_parent and item.append_daughter should normally be used for tree building. Inserting using before and after within daughters is perfectly safe.

`(item.append_daughter PARENT DAUGHTER)'
Append DAUGHTER, an item or a description of an item to the item PARENT in the PARENT's relation.

`(item.insert_parent DAUGHTER NEWPARENT)'
Insert a new parent above DAUGHTER. NEWPARENT may be a item or the description of an item.

`(item.delete ITEM)'
Delete this item from all relations it is in. All daughters of this item in each relations are also removed from the relation (which may in turn cause them to be deleted if they cease to be referenced by any other relation.

`(item.relation.remove ITEM)'
Remove this item from this relation, and any of its daughters. Other relations this item are in remain untouched.

`(item.move_tree FROM TO)'
Move the item FROM to the position of TO in TO's relation. FROM will often be in the same relation as TO but that isn't necessary. The contents of TO are dereferenced. its daughters are saved then descendants of FROM are recreated under the new TO, then TO's previous daughters are derefenced. The order of this is important as FROM may be part of TO's descendants. Note that if TO is part of FROM's descendants no moving occurs and nil is returned. For example to remove all punction terminal nodes in the Syntax relation the call would be something like

```
(define (syntax_relation_punc p)
  (if (string-equal "punc" (item.feat (item.daughter2 p) "pos"))
      (item.move_tree (item.daughter1 p) p)
  (mapcar syntax_remove_punc (item.daughters p))))
```

`(item.exchange_trees ITEM1 ITEM2)'
: Exchange `ITEM1` and `ITEM2` and their descendants in `ITEM2`'s relation. If `ITEM1` is within `ITEM2`'s descendents or vice versa `nil` is returns and no exchange takes place. If `ITEM1` is not in `ITEM2`'s relation, no exchange takes place.

Daughters of a node are actually represented as a list whose first daughter is double linked to the parent. Although being aware of this structure may be useful it is recommended that all access go through the tree specific functions `*.parent` and `*.daughter*` which properly deal with the structure, thus is the internal structure ever changes in the future only these tree access function need be updated.

With the above functions quite elaborate utterance manipulations can be performed. For example in post-lexical rules where modifications to the segments are required based on the words and their context. See section 13.8 Post-lexical rules for an example of using various utterance access functions.

14.6 Features

In previous versions items had a number of predefined features. This is no longer the case and all features are optional. Particularly the `start` and `end` features are no longer fixed, though those names are still used in the relations where yjeu are appropriate. Specific functions are provided for the `name` feature but they are just short hand for normal feature access. Simple features directly access the features in the underlying `EST_Feature` class in an item.

In addition to simple features there is a mechanism for relating functions to names, thus accessing a feature may actually call a function. For example the features `num_syls` is defined as a feature function which will count the number of syllables in the given word, rather than simple access a pre-existing feature. Feature functions are usually dependent on the particular realtion the item is in, e.g. some feature functions are only appropriate for items in the `Word` relation, or only appropriate for those in the `IntEvent` relation.

The third aspect of feature names is a path component. These are parts of the name (preceding in `.`) that indicated some trversal of the utterance structure. For example the features `name` will access the name feature on the given item. The feature `n.name` will return the name feature on the next item (in that item's relation). A number of basic direction operators are defined.

`n.`
: next

`p.`
: previous

`nn.`
: next next

`pp.`
: previous

`parent.`

`daughter1.`
: first daughter

`daughter2.`
: second daughter

`daughtern.`
: last daughter

`first.`
: most previous item

`last.`
: most next item

Also you may specific traversal to another relation relation, though the `R:<relationame>.` operator. For example given an Item in the syllable relation `R:SylStructure.parent.name` would give the name of word

the syllable is in.

Some more complex examples are as follows, assuming we are starting form an item in the `Syllable` relation.

`stress'
: This item's lexical stress

`n.stress'
: The next syllable's lexical stress

`p.stress'
: The previous syllable's lexical stress

`R:SylStructure.parent.name'
: The word this syllable is in

`R:SylStructure.parent.R:Word.n.name'
: The word next to the word this syllable is in

`n.R:SylStructure.parent.name'
: The word the next syllable is in

`R:SylStructure.daughtern.ph_vc'
: The phonetic feature `vc` of the final segment in this syllable.

A list of all feature functions is given in an appendix of this document. See section 32 Feature functions. New functions may also be added in Lisp.

In C++ feature values are of class *EST_Val* which may be a string, int, or a float (or any arbitrary object). In Scheme this distinction cannot not always be made and sometimes when you expect an int you actually get a string. Care should be take to ensure the right matching functions are use in Scheme. It is recommended you use `string-append` or `string-match` as they will always work.

If a pathname does not identify a valid path for the particular item (e.g. there is no next) `"0"` is returned.

When collecting data from speech databases it is often useful to collect a whole set of features from all utterances in a database. These features can then be used for building various models (both CART tree models and linear regression modules use these feature names),

A number of functions exist to help in this task. For example

```
(utt.features utt1 'Word '(name pos p.pos n.pos))
```

will return a list of word, and part of speech context for each word in the utterance.

See section 26.2 Extracting features for an example of extracting sets of features from a database for use in building stochastic models.

14.7 Utterance I/O

A number of functions are available to allow an utterance's structure to be made available for other programs.

The whole structure, all relations, items and features may be saved in an ascii format using the function `utt.save`. This file may be reloaded using the `utt.load` function. Note the waveform is not saved using the form.

Individual aspects of an utterance may be selectively saved. The waveform itself may be saved using the function `utt.save.wave`. This will save the waveform in the named file in the format specified in the `Parameter Wavefiletype`. All formats supported by the Edinburgh Speech Tools are valid including `nist`, `esps`, `sun`, `riff`, `aiff`, `raw` and `ulaw`. Note the functions `utt.wave.rescale` and `utt.wave.resample` may be used to change the gain and sample frequency of the waveform before saving it. A waveform may be imported into an existing utterance with the function `utt.import.wave`. This is specifically designed to allow external methods of waveform synthesis. However if you just wish to play an external wave or make it into an utterance you should consider the utterance `Wave` type.

The segments of an utterance may be saved in a file using the function `utt.save.segs` which saves the segments of the named utterance in xlabel format. Any other stream may also be saved using the more general `utt.save.relation` which takes the additional argument of a relation name. The names of each item and the end feature of each item are saved in the named file, again in Xlabel format, other features are saved in extra fields. For more elaborated saving methods you can easily write a Scheme function to save data in an utterance in whatever format is required. See the file `` `lib/mbrola.scm' `` for an example.

A simple function to allow the displaying of an utterance in Entropic's Xwaves tool is provided by the function `display`. It simply saves the waveform and the segments and sends appropriate commands to (the already running) Xwaves and xlabel programs.

A function to synthesize an externally specified utterance is provided for by `utt.resynth` which takes two filename arguments, an xlabel segment file and an F0 file. This function loads, synthesizes and plays an utterance synthesized from these files. The loading is provided by the underlying function `utt.load.segf0`.

Go to the first, previous, next, last section, table of contents.