

Festival Speech Synthesis System

This file documents the *Festival Speech Synthesis System* 1.4.3. This document contains many gaps and is still in the process of being written.

- [1. Abstract](#) initial comments
- [2. Copying](#) How you can copy and share the code
- [3. Acknowledgements](#) List of contributors
- [4. What is new](#) Enhancements since last public release

- [5. Overview](#) Generalities and Philosophy
- [6. Installation](#) Compilation and Installation
- [7. Quick start](#) Just tell me what to type
- [8. Scheme](#) A quick introduction to Festival's scripting language

Text methods for interfacing to Festival

- [9. TTS](#) Text to speech modes
- [10. XML/SGML mark-up](#) XML/SGML mark-up Language
- [11. Emacs interface](#) Using Festival within Emacs

Internal functions

- [12. Phonesets](#) Defining and using phonesets
- [13. Lexicons](#) Building and compiling Lexicons
- [14. Utterances](#) Existing and defining new utterance types

Modules

- [15. Text analysis](#) Tokenizing text
- [16. POS tagging](#) Part of speech tagging
- [17. Phrase breaks](#) Finding phrase breaks
- [18. Intonation](#) Intonations modules
- [19. Duration](#) Duration modules
- [20. UniSyn synthesizer](#) The UniSyn waveform synthesizer
- [21. Diphone synthesizer](#) Building and using diphone synthesizers
- [22. Other synthesis methods](#) other waveform synthesis methods
- [23. Audio output](#) Getting sound from Festival

- [24. Voices](#) Adding new voices (and languages)

- [25. Tools](#) CART, Ngrams etc

- [26. Building models from databases](#)

Adding new modules and writing C++ code

- [27. Programming](#) Programming in Festival (Lisp/C/C++)
- [28. API](#) Using Festival in other programs

- [29. Examples](#) Some simple (and not so simple) examples

- [30. Problems](#) Reporting bugs.
- [31. References](#) Other sources of information
- [32. Feature functions](#) List of builtin feature functions.

33. Variable list	Short descriptions of all variables
34. Function list	Short descriptions of all functions
Index	Index of concepts.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

1. Abstract

This document provides a user manual for the Festival Speech Synthesis System, version 1.4.3.

Festival offers a general framework for building speech synthesis systems as well as including examples of various modules. As a whole it offers full text to speech through a number APIs: from shell level, though a Scheme command interpreter, as a C++ library, and an Emacs interface. Festival is multi-lingual, we have developed voices in many languages including English (UK and US), Spanish and Welsh, though English is the most advanced.

The system is written in C++ and uses the Edinburgh Speech Tools for low level architecture and has a Scheme (SIOD) based command interpreter for control. Documentation is given in the FSF texinfo format which can generate a printed manual, info files and HTML.

The latest details and a full software distribution of the Festival Speech Synthesis System are available through its home page which may be found at

<http://www.cstr.ed.ac.uk/projects/festival.html>

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

2. Copying

As we feel the core system has reached an acceptable level of maturity from 1.4.0 the basic system is released under a free licence, without the commercial restrictions we imposed on early versions. The basic system has been placed under an X11 type licence which as free licences go is pretty free. No GPL code is included in festival or the speech tools themselves (though some auxiliary files are GPL'd e.g. the Emacs mode for Festival). We have deliberately chosen a licence that should be compatible with our commercial partners and our free software users.

However although the code is free, we still offer no warranties and no maintenance. We will continue to endeavor to fix bugs and answer queries when can, but are not in a position to guarantee it. We will consider maintenance contracts and consultancy if desired, please contacts us for details.

Also note that not all the voices and lexicons we distribute with festival are free. Particularly the British English lexicon derived from Oxford Advanced Learners' Dictionary is free only for non-commercial use (we will release an alternative soon). Also the Spanish diphone voice we relase is only free for non-commercial use.

If you are using Festival or the speech tools in commercial environment, even though no licence is required, we would be grateful if you let us know as it helps justify ourselves to our various sponsors.

The current copyright on the core system is

The Festival Speech Synthesis System: version 1.4.3
Centre for Speech Technology Research
University of Edinburgh, UK
Copyright (c) 1996-2002
All Rights Reserved.

Permission is hereby granted, free of charge, to use and distribute this software and its documentation without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of this work, and to permit persons to whom this work is furnished to do so, subject to the following conditions:

1. The code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Any modifications must be clearly marked as such.
3. Original authors' names are not deleted.
4. The authors' names are not used to endorse or promote products derived from this software without specific prior written permission.

THE UNIVERSITY OF EDINBURGH AND THE CONTRIBUTORS TO THIS WORK DISCLAIM ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL THE UNIVERSITY OF EDINBURGH NOR THE CONTRIBUTORS BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

3. Acknowledgements

The code in this system was primarily written by Alan W Black, Paul Taylor and Richard Caley. Festival sits on top of the Edinburgh Speech Tools Library, and uses much of its functionality.

Amy Isard wrote a synthesizer for her MSc project in 1995, which first used the Edinburgh Speech Tools Library. Although Festival doesn't contain any code from that system, her system was used as a basic model.

Much of the design and philosophy of Festival has been built on the experience both Paul and Alan gained from the development of various previous synthesizers and software systems, especially CSTR's Osprey and Polyglot systems *taylor91* and ATR's CHATR system *black94*.

However, it should be stated that Festival is fully developed at CSTR and contains neither proprietary code or ideas.

Festival contains a number of subsystems integrated from other sources and we acknowledge those systems here.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

3.1 SIOD

The Scheme interpreter (SIOD -- Scheme In One Defun 3.0) was written by George Carrett (gjc@mitech.com, gjc@paradigm.com) and offers a basic small Scheme (Lisp) interpreter suitable for embedding in applications such

as Festival as a scripting language. A number of changes and improvements have been added in our development but it still remains that basic system. We are grateful to George and Paradigm Associates Incorporated for providing such a useful and well-written sub-system.

Scheme In One Defun (SIOD)
COPYRIGHT (c) 1988-1994 BY
PARADIGM ASSOCIATES INCORPORATED, CAMBRIDGE, MASSACHUSETTS.
ALL RIGHTS RESERVED

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Paradigm Associates Inc not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

PARADIGM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL PARADIGM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

3.2 editline

Because of conflicts between the copyright for GNU readline, for which an optional interface was included in earlier versions, we have replace the interface with a complete command line editing system based on `editline'. `Editline' was posted to the USENET newsgroup `comp.sources.misc' in 1992. A number of modifications have been made to make it more useful to us but the original code (contained within the standard speech tools distribution) and our modifications fall under the following licence.

Copyright 1992 Simmule Turner and Rich Salz. All rights reserved.

This software is not subject to any license of the American Telephone and Telegraph Company or of the Regents of the University of California.

Permission is granted to anyone to use this software for any purpose on any computer system, and to alter it and redistribute it freely, subject to the following restrictions:

1. The authors are not responsible for the consequences of use of this software, no matter how awful, even if they arise from flaws in it.
2. The origin of this software must not be misrepresented, either by explicit claim or by omission. Since few users ever read sources, credits must appear in the documentation.
3. Altered versions must be plainly marked as such, and must not be misrepresented as being the original software. Since few users ever read sources, credits must appear in the documentation.
4. This notice may not be removed or altered.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

3.3 Edinburgh Speech Tools Library

The Edinburgh Speech Tools lies at the core of Festival. Although developed separately, much of the development of certain parts of the Edinburgh Speech Tools has been directed by Festival's needs. In turn those who have contributed to the Speech Tools make Festival a more usable system.

See section 'Acknowledgements' in *Edinburgh Speech Tools Library Manual*.

Online information about the Edinburgh Speech Tools library is available through

http://www.cstr.ed.ac.uk/projects/speech_tools.html

[<] [>] [Up] [>>] [Top] [Contents] [Index] [?]

3.4 Others

Many others have provided actual code and support for Festival, for which we are grateful. Specifically,

- Alistair Conkie: various low level code points and some design work, Spanish synthesis, the old diphone synthesis code.
- Steve Isard: directorship and LPC diphone code, design of diphone schema.
- EPSRC: who fund Alan Black and Paul Taylor.
- Sun Microsystems Laboratories: for supporting the project and funding Richard.
- AT&T Labs - Research: for supporting the project.
- Paradigm Associates and George Carrett: for Scheme in one defun.
- Mike Macon: Improving the quality of the diphone synthesizer and LPC analysis.
- Kurt Dusterhoff: Tilt intonation training and modelling.
- Amy Isard: for her SSML project and related synthesizer.
- Richard Tobin: for answering all those difficult questions, the socket code, and the XML parser.
- Simmule Turner and Rich Salz: command line editor (editline)
- Borja Etxebarria: Help with the Spanish synthesis
- Briony Williams: Welsh synthesis
- Jacques H. de Villiers: `jacques@cse.ogi.edu' from CSLU at OGI, for the TCL interface, and other usability issues
- Kevin Lenzo: `lenzo@cs.cmu.edu' from CMU for the PERL interface.
- Rob Clarke: for support under Linux.
- Samuel Audet `guardia@cam.org': OS/2 support
- Mari Ostendorf: For providing access to the BU FM Radio corpus from which some modules were trained.
- Melvin Hunt: from whose work we based our residual LPC synthesis model on
- Oxford Text Archive: For the computer users version of Oxford Advanced Learners' Dictionary (redistributed with permission).
- Reading University: for access to MARSEC from which the phrase break model was trained.
- LDC & Penn Tree Bank: from which the POS tagger was trained, redistribution of the models is with permission from the LDC.
- Roger Burroughes and Kurt Dusterhoff: For letting us capture their voices.
- ATR and Nick Campbell: for first getting Paul and Alan to work together and for the experience we gained.
- FSF: for G++, make,
- Center for Spoken Language Understanding: CSLU at OGI, particularly Ron Cole and Mike Macon, have acted as significant users for the system giving significant feedback and allowing us to teach courses on Festival offering valuable real-use feedback.
- Our beta testers: Thanks to all the people who put up with previous versions of the system and reported bugs, both big and small. These comments are very important to the constant improvements in the system. And thanks for your quick responses when I had specific requests.
- And our users ... Many people have downloaded earlier versions of the system. Many have found problems with installation and use and have reported it to us. Many of you have put up with multiple compilations

trying to fix bugs remotely. We thank you for putting up with us and are pleased you've taken the time to help us improve our system. Many of you have come up with uses we hadn't thought of, which is always rewarding.

Even if you haven't actively responded, the fact that you use the system at all makes it worthwhile.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

4. What is new

Compared to the the previous major release (1.3.0 release Aug 1998) 1.4.0 is not functionally so different from its previous versions. This release is primarily a consolidation release fixing and tidying up some of the lower level aspects of the system to allow better modularity for some of our future planned modules.

- Copyright change: The system is now free and has no commercial restriction. Note that currently on the US voices (ked and kal) are also now unrestricted. The UK English voices depend on the Oxford Advanced Learners' Dictionary of Current English which cannot be used for commercial use without permission from Oxford University Press.
- Architecture tidy up: the interfaces to lower level part parts of the system have been tidied up deleting some of the older code that was supported for compatibility reasons. This is a much higher dependence of features and easier (and safer) ways to register new objects as feature values and Scheme objects. Scheme has been tidied up. It is no longer "in one defun" but "in one directory".
- New documentation system for speech tools: A new docbook based documentation system has been added to the speech tools. Festival's documentation will will move over to this sometime soon too.
- initial JSAPI support: both JSAPI and JSML (somewhat similar to Sable) now have initial impelementsations. They of course depend on Java support which so far we have only (successfully) investgated under Solaris and Linux.
- Generalization of statistical models: CART, ngrams, and WFSTs are now fully supported from Lisp and can be used with a generalized viterbi function. This makes adding quite complex statistical models easy without adding new C++.
- Tilt Intonation modelling: Full support is now included for the Tilt intomation models, both training and use.
- Documentation on Bulding New Voices in Festival: documentation, scripts etc. for building new voices and languages in the system, see <http://www.cstr.ed.ac.uk/projects/festival/docs/festvox/>

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

5. Overview

Festival is designed as a speech synthesis system for at least three levels of user. First, those who simply want high quality speech from arbitrary text with the minimum of effort. Second, those who are developing language systems and wish to include synthesis output. In this case, a certain amount of customization is desired, such as different voices, specific phrasing, dialog types etc. The third level is in developing and testing new synthesis methods.

This manual is not designed as a tutorial on converting text to speech but for documenting the processes and use of our system. We do not discuss the detailed algorithms involved in converting text to speech or the relative merits of

multiple methods, though we will often give references to relevant papers when describing the use of each module.

For more general information about text to speech we recommend Dutoit's 'An introduction to Text-to-Speech Synthesis' [dutoit97](#). For more detailed research issues in TTS see [sproat98](#) or [vansanten96](#).

[5.1 Philosophy](#) Why we did it like it is

[5.2 Future](#) How much better its going to get

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

5.1 Philosophy

One of the biggest problems in the development of speech synthesis, and other areas of speech and language processing systems, is that there are a lot of simple well-known techniques lying around which can help you realise your goal. But in order to improve some part of the whole system it is necessary to have a whole system in which you can test and improve your part. Festival is intended as that whole system in which you may simply work on your small part to improve the whole. Without a system like Festival, before you could even start to test your new module you would need to spend significant effort to build a whole system, or adapt an existing one before you could start working on your improvements.

Festival is specifically designed to allow the addition of new modules, easily and efficiently, so that development need not get bogged down in re-implementing the wheel.

But there is another aspect of Festival which makes it more useful than simply an environment for researching into new synthesis techniques. It is a fully usable text-to-speech system suitable for embedding in other projects that require speech output. The provision of a fully working easy-to-use speech synthesizer in addition to just a testing environment is good for two specific reasons. First, it offers a conduit for our research, in that our experiments can quickly and directly benefit users of our synthesis system. And secondly, in ensuring we have a fully working usable system we can immediately see what problems exist and where our research should be directed rather where our whims take us.

These concepts are not unique to Festival. ATR's CHATR system ([black94](#)) follows very much the same philosophy and Festival benefits from the experiences gained in the development of that system. Festival benefits from various pieces of previous work. As well as CHATR, CSTR's previous synthesizers, Osprey and the Polyglot projects influenced many design decisions. Also we are influenced by more general programs in considering software engineering issues, especially GNU Octave and Emacs on which the basic script model was based.

Unlike in some other speech and language systems, software engineering is considered very important to the development of Festival. Too often research systems consist of random collections of hacky little scripts and code. No one person can confidently describe the algorithms it performs, as parameters are scattered throughout the system, with tricks and hacks making it impossible to really evaluate why the system is good (or bad). Such systems do not help the advancement of speech technology, except perhaps in pointing at ideas that should be further investigated. If the algorithms and techniques cannot be described externally from the program *such that* they can reimplemented by others, what is the point of doing the work?

Festival offers a common framework where multiple techniques may be implemented (by the same or different researchers) so that they may be tested more fairly in the same environment.

As a final word, we'd like to make two short statements which both achieve the same end but unfortunately perhaps not for the same reasons:

Good software engineering makes good research easier

But the following seems to be true also

If you spend enough effort on something it can be shown to be better than its competitors.

5.2 Future

Festival is still very much in development. Hopefully this state will continue for a long time. It is never possible to complete software, there are always new things that can make it better. However as time goes on Festival's core architecture will stabilise and little or no changes will be made. Other aspects of the system will gain greater attention such as waveform synthesis modules, intonation techniques, text type dependent analysers etc.

Festival will improve, so don't expect it to be the same six months from now.

A number of new modules and enhancements are already under consideration at various stages of implementation. The following is a non-exhaustive list of what we may (or may not) add to Festival over the next six months or so.

- Selection-based synthesis: Moving away from diphone technology to more generalized selection of units for speech database.
- New structure for linguistic content of utterances: Using techniques for Metrical Phonology we are building more structure representations of utterances reflecting their linguistic significance better. This will allow improvements in prosody and unit selection.
- Non-prosodic prosodic control: For language generation systems and custom tasks where the speech to be synthesized is being generated by some program, more information about text structure will probably exist, such as phrasing, contrast, key items etc. We are investigating the relationship of high-level tags to prosodic information through the Sole project <http://www.cstr.ed.ac.uk/projects/sole.html>
- Dialect independent lexicons: Currently for each new dialect we need a new lexicon, we are currently investigating a form of lexical specification that is dialect independent that allows the core form to be mapped to different dialects. This will make the generation of voices in different dialects much easier.

6. Installation

This section describes how to install Festival from source in a new location and customize that installation.

6.1 Requirements	Software/Hardware requirements for Festival
6.2 Configuration	Setting up compilation
6.3 Site initialization	Settings for your particular site
6.4 Checking an installation	But does it work ...

6.1 Requirements

In order to compile Festival you first need the following source packages

```
festival-1.4.3-release.tar.gz
    Festival Speech Synthesis System source
speech_tools-1.2.3-release.tar.gz
    The Edinburgh Speech Tools Library
festlex_NAME.tar.gz
```

The lexicon distribution, where possible, includes the lexicon input file as well as the compiled form, for your convenience. The lexicons have varying distribution policies, but are all free except OALD, which is only free

for non-commercial use (we are working on a free replacement). In some cases only a pointer to an ftp'able file plus a program to convert that file to the Festival format is included.

`festvox_NAME.tar.gz`

You'll need a speech database. A number are available (with varying distribution policies). Each voice may have other dependencies such as requiring particular lexicons

`festdoc_1.4.3.tar.gz`

Full postscript, info and html documentation for Festival and the Speech Tools. The source of the documentation is available in the standard distributions but for your conveniences it has been pre-generated.

In addition to Festival specific sources you will also need

A UNIX machine

Currently we have compiled and tested the system under Solaris (2.5(.1), 2.6, 2.7 and 2.8), SunOS (4.1.3), FreeBSD 3.x, 4.x Linux (Redhat 4.1, 5.0, 5.1, 5.2, 6.[012], 7.[01], 8.0 and other Linux distributions), and it should work under OSF (Dec Alphas) SGI (Irix), HPs (HPUX). But any standard UNIX machine should be acceptable. We have now successfully ported this version to Windows NT nad Windows 95 (using the Cygnus GNU win32 environment). This is still a young port but seems to work.

A C++ compiler

Note that C++ is not very portable even between different versions of the compiler from the same vendor. Although we've tried very hard to make the system portable, we know it is very unlikely to compile without change except with compilers that have already been tested. The currently tested systems are

- o Sun Sparc Solaris 2.5, 2.5.1, 2.6, 2.7, 2.9: GCC 2.95.1, GCC 3.2
- o FreeBSD for Intel 3.x and 4.x GCC 2.95.1, GCC 3.0
- o Linux for Intel (RedHat 4.1/5.0/5.1/5.2/6.0/7.x/8.0): GCC 2.7.2, GCC 2.7.2/egcs-1.0.2, egcs 1.1.1, egcs-1.1.2, GCC 2.95.[123], GCC "2.96", GCC 3.0, GCC 3.0.1 GCC 3.2 GCC 3.2.1
- o Windows NT 4.0: GCC 2.7.2 plus egcs (from Cygnus GNU win32 b19), Visual C++ PRO v5.0, Visual C++ v6.0

Note if GCC works on one version of Unix it usually works on others.

We have compiled both the speech tools and Festival under Windows NT 4.0 and Windows 95 using the GNU tools available from Cygnus.

<ftp://ftp.cygnus.com/pub/gnu-win32/>.

GNU make

Due to there being too many different make programs out there we have tested the system using GNU make on all systems we use. Others may work but we know GNU make does.

Audio hardware

You can use Festival without audio output hardware but it doesn't sound very good (though admittedly you can hear less problems with it). A number of audio systems are supported (directly inherited from the audio support in the Edinburgh Speech Tools Library): NCD's NAS (formerly called netaudio) a network transparent audio system (which can be found at <ftp://ftp.x.org/contrib/audio/nas/>); ``/dev/audio'` (at 8k ulaw and 8/16bit linear), found on Suns, Linux machines and FreeBSD; and a method allowing arbitrary UNIX commands. See section [23. Audio output](#).

Earlier versions of Festival mistakenly offered a command line editor interface to the GNU package readline, but due to conflicts with the GNU Public Licence and Festival's licence this interface was removed in version 1.3.1. Even Festival's new free licence would cause problems as readline support would restrict Festival linking with non-free code. A new command line interface based on editline was provided that offers similar functionality. Editline remains a compilation option as it is probably not yet as portable as we would like it to be.

In addition to the above, in order to process the documentation you will need ``TeX'`, ``dvips'` (or similar), GNU's ``makeinfo'` (part of the texinfo package) and ``texi2html'` which is available from <http://wwwcn.cern.ch/dci/texi2html/>.

However the document files are also available pre-processed into, postscript, DVI, info and html as part of the distribution in ``festdoc-1.4.X.tar.gz'`.

Ensure you have a fully installed and working version of your C++ compiler. Most of the problems people have had in installing Festival have been due to incomplete or bad compiler installation. It might be worth checking if the following program works if you don't know if anyone has used your C++ installation before.

```
#include <iostream.h>
int main (int argc, char **argv)
{
    cout << "Hello world\n";
}
```

Unpack all the source files in a new directory. The directory will then contain two subdirectories

```
speech_tools/
festival/
```

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

6.2 Configuration

First ensure you have a compiled version of the Edinburgh Speech Tools Library. See ``speech_tools/INSTALL'` for instructions.

The system now supports the standard GNU ``configure'` method for set up. In most cases this will automatically configure festival for your particular system. In most cases you need only type

```
gmake
```

and the system will configure itself and compile, (note you need to have compiled the Edinburgh Speech Tools ``speech_tools-1.2.2'` first.

In some case hand configure is require. All of the configuration choise are held in the file ``config/config'`

For the most part Festival configuration inherits the configuration from your speech tools config file (``../speech_tools/config/config'`). Additional optional modules may be added by adding them to the end of your config file e.g.

```
ALSO_INCLUDE += clunits
```

Adding and new module here will treat is as a new directory in the ``src/modules/'` and compile it into the system in the same way the `OTHER_DIRS` feature was used in previous versions.

If the compilation directory being accessed by NFS or if you use an automounter (e.g. amd) it is recommend to explicitly set the variable `FESTIVAL_HOME` in ``config/config'`. The command `pwd` is not reliable when a directory may have multiple names.

There is a simple test suite with Festival but it requires the three basic voices and their respective lexicons install before it will work. Thus you need to install

```
festlex_CMU.tar.gz
festlex_OALD.tar.gz
festlex_POSLEX.tar.gz
festvox_don.tar.gz
festvox_kedlpc16k.tar.gz
festvox_rablpc16k.tar.gz
```

If these are installed you can test the installation with
`gmake test`

To simply make it run with a male US English voice it is sufficient to install just

```
festlex_CMU.tar.gz
festlex_POSLEX.tar.gz
festvox_kallpc16k.tar.gz
```

Note that the single most common reason for problems in compilation and linking found amongst the beta testers was a bad installation of GNU C++. If you get many strange errors in G++ library header files or link errors it is worth checking that your system has the compiler, header files and runtime libraries properly installed. This may be checked by compiling a simple program under C++ and also finding out if anyone at your site has ever used the installation. Most of these installation problems are caused by upgrading to a newer version of libg++ without removing the older version so a mixed version of the `.h` files exist.

Although we have tried very hard to ensure that Festival compiles with no warnings this is not possible under some systems.

Under SunOS the system include files do not declare a number of system provided functions. This a bug in Sun's include files. This will cause warnings like "implicit definition of fprintf". These are harmless.

Under Linux a warning at link time about reducing the size of some symbols often is produced. This is harmless. There is often occasional warnings about some socket system function having an incorrect argument type, this is also harmless.

The speech tools and festival compile under Windows95 or Windows NT with Visual C++ v5.0 using the Microsoft `nmake` make program. We've only done this with the Professional edition, but have no reason to believe that it relies on anything not in the standard edition.

In accordance to VC++ conventions, object files are created with extension `.obj`, executables with extension `.exe` and libraries with extension `.lib`. This may mean that both unix and Win32 versions can be built in the same directory tree, but I wouldn't rely on it.

To do this you require `nmake` Makefiles for the system. These can be generated from the `gnumake` Makefiles, using the command

```
gnumake VCMakefile
```

in the `speech_tools` and `festival` directories. I have only done this under unix, it's possible it would work under the `gnuwin32` system.

If `make.depend` files exist (i.e. if you have done `gnumake depend` in unix) equivalent `vc_make.depend` files will be created, if not the `VCMakefiles` will not contain dependency information for the `.cc` files. The result will be that you can compile the system once, but changes will not cause the correct things to be rebuilt.

In order to compile from the DOS command line using Visual C++ you need to have a collection of environment variables set. In Windows NT there is an installation option for Visual C++ which sets these globally. Under

Windows95 or if you don't ask for them to be set globally under NT you need to run

```
vcvars32.bat
```

See the VC++ documentation for more details.

Once you have the source trees with VCMakefiles somewhere visible from Windows, you need to copy ``peech_tools\config\vc_config-dist'` to ``speech_tools\config\vc_config'` and edit it to suit your local situation. Then do the same with ``festival\config\vc_config-dist'`.

The thing most likely to need changing is the definition of `FESTIVAL_HOME` in ``festival\config\vc_config_make_rules'` which needs to point to where you have put festival.

Now you can compile. cd to the `speech_tools` directory and do

```
nmake /nologo /fVCMakefile
```

and the library, the programs in main and the test programs should be compiled.

The tests can't be run automatically under Windows. A simple test to check that things are probably OK is:

```
main\na_play testsuite\data\ch_wave.wav
```

which reads and plays a waveform.

Next go into the `festival` directory and do

```
nmake /nologo /fVCMakefile
```

to build festival. When it's finished, and assuming you have the voices and lexicons unpacked in the right place, festival should run just as under unix.

We should remind you that the NT/95 ports are still young and there may yet be problems that we've not found yet. We only recommend the use the speech tools and Festival under Windows if you have significant experience in C++ under those platforms.

Most of the modules ``src/modules'` are actually optional and the system could be compiled without them. The basic set could be reduced further if certain facilities are not desired. Particularly: ``donovan'` which is only required if the donovan voice is used; ``rxp'` if no XML parsing is required (e.g. Sable); and ``parser'` if no stochastic paring is required (this parser isn't used for any of our currently released voices). Actually even ``UniSyn'` and ``UniSyn_diphone'` could be removed if some external waveform synthesizer is being used (e.g. MBROLA) or some alternative one like ``OGIresLPC'`. Removing unused modules will make the festival binary smaller and (potentially) start up faster but don't expect too much. You can delete these by changing the `BASE_DIRS` variable in ``src/modules/Makefile'`.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

6.3 Site initialization

Once compiled Festival may be further customized for particular sites. At start up time Festival loads the file ``init.scm'` from its library directory. This file further loads other necessary files such as phoneset descriptions, duration parameters, intonation parameters, definitions of voices etc. It will also load the files ``sitevars.scm'` and ``siteinit.scm'` if they exist. ``sitevars.scm'` is loaded after the basic Scheme library functions are loaded but before any of the festival related functions are loaded. This file is intended to set various path names before various subsystems are loaded. Typically variables such as `lexdir` (the directory where the lexicons are

held), and `voices_dir` (pointing to voice directories) should be reset here if necessary.

The default installation will try to find its lexicons and voices automatically based on the value of `load-path` (this is derived from `FESTIVAL_HOME` at compilation time or by using the `--libdir` at run-time). If the voices and lexicons have been unpacked into subdirectories of the library directory (the default) then no site specific initialization of the above pathnames will be necessary.

The second site specific file is ``siteinit.scm'`. Typical examples of local initialization are as follows. The default audio output method is NCD's NAS system if that is supported as that's what we use normally in CSTR. If it is not supported, any hardware specific mode is the default (e.g. `sun16audio`, `freebas16audio`, `linux16audio` or `mplayeraudio`). But that default is just a setting in ``init.scm'`. If for example in your environment you may wish the default audio output method to be 8k mulaw through ``/dev/audio'` you should add the following line to your ``siteinit.scm'` file

```
(Parameter.set 'Audio_Method 'sunaudio)
```

Note the use of `Parameter.set` rather than `Parameter.def` the second function will not reset the value if it is already set. Remember that you may use the audio methods `sun16audio`, `linux16audio` or `freebsd16audio` only if `NATIVE_AUDIO` was selected in ``speech_tools/config/config'` and you are on such machines. The Festival variable `*modules*` contains a list of all supported functions/modules in a particular installation including audio support. Check the value of that variable if things aren't what you expect.

If you are installing on a machine whose audio is not directly supported by the speech tools library, an external command may be executed to play a waveform. The following example is for an imaginary machine that can play audio files through a program called ``adplay'` with arguments for sample rate and file type. When playing waveforms, Festival, by default, outputs as unheadered waveform in native byte order. In this example you would set up the default audio playing mechanism in ``siteinit.scm'` as follows

```
(Parameter.set 'Audio_Method 'Audio_Command)
(Parameter.set 'Audio_Command "adplay -raw -r $SR $FILE")
```

For `Audio_Command` method of playing waveforms Festival supports two additional audio parameters. `Audio_Required_Rate` allows you to use Festival's internal sample rate conversion function to any desired rate. Note this may not be as good as playing the waveform at the sample rate it is originally created in, but as some hardware devices are restrictive in what sample rates they support, or have naive resample functions this could be optimal. The second addition audio parameter is `Audio_Required_Format` which can be used to specify the desired output forms of the file. The default is unheadered raw, but this may be any of the values supported by the speech tools (including `nist`, `esps`, `snd`, `riff`, `aiff`, `audlab`, `raw` and, if you really want it, `ascii`).

For example suppose you run Festival on a remote machine and are not running any network audio system and want Festival to copy files back to your local machine and simply cat them to ``/dev/audio'`. The following would do that (assuming permissions for `rsh` are allowed).

```
(Parameter.set 'Audio_Method 'Audio_Command)
;; Make output file ulaw 8k (format ulaw implies 8k)
(Parameter.set 'Audio_Required_Format 'ulaw)
(Parameter.set 'Audio_Command
"userhost=`echo $DISPLAY | sed 's/:.*$//'\`; rcp $FILE $userhost:$FILE; \
rsh $userhost \"cat $FILE >/dev/audio\" ; rsh $userhost \"rm $FILE\"")
```

Note there are limits on how complex a command you want to put in the `Audio_Command` string directly. It can get very confusing with respect to quoting. It is therefore recommended that once you get past a certain complexity consider writing a simple shell script and calling it from the `Audio_Command` string.

A second typical customization is setting the default speaker. Speakers depend on many things but due to various licence (and resource) restrictions you may only have some diphone/nphone databases available in your installation. The function name that is the value of `voice_default` is called immediately after ``siteinit.scm'` is loaded offering the opportunity for you to change it. In the standard distribution no change should be required. If you download all the distributed voices `voice_rab_diphone` is the default voice. You may change this for a site by

adding the following to ``siteinit.scm'` or per person by changing your ``.festivalrc'`. For example if you wish to change the default voice to the American one `voice_ked_diphone`

```
(set! voice_default 'voice_ked_diphone)
```

Note the single quote, and note that unlike in early versions `voice_default` is not a function you can call directly.

A second level of customization is on a per user basis. After loading ``init.scm'`, which includes ``sitevars.scm'` and ``siteinit.scm'` for local installation, Festival loads the file ``.festivalrc'` from the user's home directory (if it exists). This file may contain arbitrary Festival commands.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

6.4 Checking an installation

Once compiled and site initialization is set up you should test to see if Festival can speak or not.

Start the system

```
$ bin/festival
Festival Speech Synthesis System 1.4.3:release Jan 2003
Copyright (C) University of Edinburgh, 1996-2003. All rights reserved.
For details type `(festival_warranty)'
festival> ^D
```

If errors occur at this stage they are most likely to do with pathname problems. If any error messages are printed about non-existent files check that those pathnames point to where you intended them to be. Most of the (default) pathnames are dependent on the basic library path. Ensure that is correct. To find out what it has been set to, start the system without loading the init files.

```
$ bin/festival -q
Festival Speech Synthesis System 1.4.3:release Jan 2003
Copyright (C) University of Edinburgh, 1996-2003. All rights reserved.
For details type `(festival_warranty)'
festival> libdir
"/projects/festival/lib/"
festival> ^D
```

This should show the pathname you set in your ``config/config'`.

If the system starts with no errors try to synthesize something

```
festival> (SayText "hello world")
```

Some files are only accessed at synthesis time so this may show up other problem pathnames. If it talks, you're in business, if it doesn't, here are some possible problems.

If you get the error message

```
Can't access NAS server
```

You have selected NAS as the audio output but have no server running on that machine or your `DISPLAY` or `AUDIOSERVER` environment variable is not set properly for your output device. Either set these properly or change the audio output device in ``lib/siteinit.scm'` as described above.

Ensure your audio device actually works the way you think it does. On Suns, the audio output device can be switched into a number of different output modes, speaker, jack, headphones. If this is set to the wrong one you may not hear the output. Use one of Sun's tools to change this (try ``/usr/demo/SOUND/bin/soundtool``). Try to find an audio file independent of Festival and get it to play on your audio. Once you have done that ensure that the audio output method set in Festival matches that.

Once you have got it talking, test the audio spooling device.

```
festival> (intro)
```

This plays a short introduction of two sentences, spooling the audio output.

Finally exit from Festival (by end of file or `(quit)`) and test the script mode with.

```
$ examples/saytime
```

A test suite is included with Festival but it makes certain assumptions about which voices are installed. It assumes that `voice_rab_diphone`festvox_rabxxxx.tar.gz`` is the default voice and that `voice_ked_diphone` and `voice_don_diphone`festvox_kedxxxx.tar.gz`` and ``festvox_don.tar.gz`` are installed. Also local settings in your ``festival/lib/siteinit.scm`` may affect these tests. However, after installation it may be worth trying

```
gnumake test
```

from the ``festival/`` directory. This will do various tests including basic utterance tests and tokenization tests. It also checks that voices are installed and that they don't interfere with each other. These tests are primarily regression tests for the developers of Festival, to ensure new enhancements don't mess up existing supported features. They are not designed to test an installation is successful, though if they run correctly it is most probable the installation has worked.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

7. Quick start

This section is for those who just want to know the absolute basics to run the system.

Festival works in two fundamental modes, *command mode* and *text-to-speech mode* (`tts-mode`). In command mode, information (in files or through standard input) is treated as commands and is interpreted by a Scheme interpreter. In `tts-mode`, information (in files or through standard input) is treated as text to be rendered as speech. The default mode is command mode, though this may change in later versions.

[7.1 Basic command line options](#)

[7.2 Sample command driven session](#)

[7.3 Getting some help](#)

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

7.1 Basic command line options

Festival's basic calling method is as


```
festival [options] file1 file2 ...
```

Options may be any of the following

```
-q      start Festival without loading `init.scm' or user's `.festivalrc'
```

```
-b
```

```
--batch      After processing any file arguments do not become interactive
```

```
-i
```

```
--interactive  After processing file arguments become interactive. This option overrides any batch argument.
```

```
--tts        Treat file arguments in text-to-speech mode, causing them to be rendered as speech rather than interpreted as commands. When selected in interactive mode the command line edit functions are not available
```

```
--command    Treat file arguments in command mode. This is the default.
```

```
--language LANG      Set the default language to LANG. Currently LANG may be one of english, spanish or welsh (depending on what voices are actually available in your installation).
```

```
--server     After loading any specified files go into server mode. This is a mode where Festival waits for clients on a known port (the value of server_port, default is 1314). Connected clients may send commands (or text) to the server and expect waveforms back. See section 28.3 Server/client API. Note server mode may be unsafe and allow unauthorised access to your machine, be sure to read the security recommendations in 28.3 Server/client API
```

```
--script scriptfile  Run scriptfile as a Festival script file. This is similar to --batch but it encapsulates the command line arguments into the Scheme variables argv and argc, so that Festival scripts may process their command line arguments just like any other program. It also does not load the the basic initialisation files as sometimes you may not want to do this. If you wish them, you should copy the loading sequence from an example Festival script like `festival/examples/saytext'.
```

```
--heap NUMBER      The Scheme heap (basic number of Lisp cells) is of a fixed size and cannot be dynamically increased at run time (this would complicate garbage collection). The default size is 210000 which seems to be more than adequate for most work. In some of our training experiments where very large list structures are required it is necessary to increase this. Note there is a trade off between size of the heap and time it takes to garbage collect so making this unnecessarily big is not a good idea. If you don't understand the above explanation you almost certainly don't need to use the option.
```

In command mode, if the file name starts with a left parenthesis, the name itself is read and evaluated as a Lisp command. This is often convenient when running in batch mode and a simple command is necessary to start the whole thing off after loading in some other specific files.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

7.2 Sample command driven session

Here is a short session using Festival's command interpreter.

Start Festival with no arguments

```
$ festival
Festival Speech Synthesis System 1.4.3:release Dec 2002
Copyright (C) University of Edinburgh, 1996-2002. All rights reserved.
For details type `(festival_warranty)'
festival>
```

Festival uses the a command line editor based on editline for terminal input so command line editing may be done with Emacs commands. Festival also supports history as well as function, variable name, and file name completion via the TAB key.

Typing `help` will give you more information, that is `help` without any parenthesis. (It is actually a variable name whose value is a string containing `help`.)

Festival offers what is called a read-eval-print loop, because it reads an s-expression (atom or list), evaluates it and prints the result. As Festival includes the SIOD Scheme interpreter most standard Scheme commands work

```
festival> (car '(a d))
a
festival> (+ 34 52)
86
```

In addition to standard Scheme commands a number of commands specific to speech synthesis are included. Although, as we will see, there are simpler methods for getting Festival to speak, here are the basic underlying explicit functions used in synthesizing an utterance.

Utterances can consist of various types See section [14.2 Utterance types](#), but the simplest form is plain text. We can create an utterance and save it in a variable

```
festival> (set! utt1 (Utterance Text "Hello world"))
#<Utterance 1d08a0>
festival>
```

The (hex) number in the return value may be different for your installation. That is the print form for utterances. Their internal structure can be very large so only a token form is printed.

Although this creates an utterance it doesn't do anything else. To get a waveform you must synthesize it.

```
festival> (utt.synth utt1)
#<Utterance 1d08a0>
festival>
```

This calls various modules, including tokenizing, duration, intonation etc. Which modules are called are defined with respect to the type of the utterance, in this case `Text`. It is possible to individually call the modules by hand but you just wanted it to talk didn't you. So

```
festival> (utt.play utt1)
#<Utterance 1d08a0>
festival>
```

will send the synthesized waveform to your audio device. You should hear "Hello world" from your machine.

To make this all easier a small function doing these three steps exists. `SayText` simply takes a string of text, synthesizes it and sends it to the audio device.

```
festival> (SayText "Good morning, welcome to Festival")
#<Utterance 1d8fd0>
festival>
```

Of course as history and command line editing are supported `c-p` or up-arrow will allow you to edit the above to whatever you wish.

Festival may also synthesize from files rather than simply text.

```
festival> (tts "myfile" nil)
nil
festival>
```

The end of file character `c-d` will exit from Festival and return you to the shell, alternatively the command `quit` may be called (don't forget the parentheses).

Rather than starting the command interpreter, Festival may synthesize files specified on the command line

```
unix$ festival --tts myfile
unix$
```

Sometimes a simple waveform is required from text that is to be kept and played at some later time. The simplest way to do this with festival is by using the `text2wave` program. This is a festival script that will take a file (or text from standard input) and produce a single waveform.

An example use is

```
text2wave myfile.txt -o myfile.wav
```

Options exist to specify the waveform file type, for example if Sun audio format is required

```
text2wave myfile.txt -otype snd -o myfile.wav
```

Use `-h` on `text2wave` to see all options.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

7.3 Getting some help

If no audio is generated then you must check to see if audio is properly initialized on your machine. See section [23. Audio output](#).

In the command interpreter `m-h` (meta-h) will give you help on the current symbol before the cursor. This will be a short description of the function or variable, how to use it and what its arguments are. A listing of all such help strings appears at the end of this document. `m-s` will synthesize and say the same information, but this extra function is really just for show.

The lisp function `manual` will send the appropriate command to an already running Netscape browser process. If `nil` is given as an argument the browser will be directed to the tables of contents of the manual. If a non-nil value is given it is assumed to be a section title and that section is searched and if found displayed. For example

```
festival> (manual "Accessing an utterance")
```

Another related function is `manual-sym` which given a symbol will check its documentation string for a cross

reference to a manual section and request Netscape to display it. This function is bound to `m-m` and will display the appropriate section for the given symbol.

Note also that the `TAB` key can be used to find out the name of commands available as can the function `Help` (remember the parentheses).

For more up to date information on Festival regularly check the Festival Home Page at

<http://www.cstr.ed.ac.uk/projects/festival.html>

Further help is available by mailing questions to

`festival-help@cstr.ed.ac.uk`

Although we cannot guarantee the time required to answer you, we will do our best to offer help.

Bug reports should be submitted to

`festival-bug@cstr.ed.ac.uk`

If there is enough user traffic a general mailing list will be created so all users may share comments and receive announcements. In the mean time watch the Festival Home Page for news.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

8. Scheme

Many people seem daunted by the fact that Festival uses Scheme as its scripting language and feel they can't use Festival because they don't know Scheme. However most of those same people use Emacs everyday which also has (a much more complex) Lisp system underneath. The number of Scheme commands you actually need to know in Festival is really very small and you can easily just find out as you go along. Also people use the Unix shell often but only know a small fraction of actual commands available in the shell (or in fact that there even is a distinction between shell builtin commands and user definable ones). So take it easy, you'll learn the commands you need fairly quickly.

- [8.1 Scheme references](#) Places to learn more about Scheme
- [8.2 Scheme fundamentals](#) Syntax and semantics
- [8.3 Scheme Festival specifics](#)
- [8.4 Scheme I/O](#)

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

8.1 Scheme references

If you wish to learn about Scheme in more detail I recommend the book *abelson85*.

The Emacs Lisp documentation is reasonable as it is comprehensive and many of the underlying uses of Scheme in Festival were influenced by Emacs. Emacs Lisp however is not Scheme so there are some differences.

Other Scheme tutorials and resources available on the Web are

- The Revised Revised Revised Revised Scheme Report, the document defining the language is available from http://tinuviel.cs.wcu.edu/res/ldp/r4rs-html/r4rs_toc.html
- a Scheme tutorials from the net:
 - <http://www.cs.uoregon.edu/classes/cis425/schemeTutorial.html>
- the Scheme FAQ
 - <http://www.landfield.com/faqs/scheme-faq/part1/>

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

8.2 Scheme fundamentals

But you want more now, don't you, not just be referred to some other book. OK here goes.

Syntax: an expression is an *atom* or a *list*. A list consists of a left paren, a number of expressions and right paren. Atoms can be symbols, numbers, strings or other special types like functions, hash tables, arrays, etc.

Semantics: All expressions can be evaluated. Lists are evaluated as function calls. When evaluating a list all the members of the list are evaluated first then the first item (a function) is called with the remaining items in the list as arguments. Atoms are evaluated depending on their type: symbols are evaluated as variables returning their values. Numbers, strings, functions, etc. evaluate to themselves.

Comments are started by a semicolon and run until end of line.

And that's it. There is nothing more to the language that. But just in case you can't follow the consequences of that, here are some key examples.

```
festival> (+ 2 3)
5
festival> (set! a 4)
4
festival> (* 3 a)
12
festival> (define (add a b) (+ a b))
#<CLOSURE (a b) (+ a b)>
festival> (add 3 4)
7
festival> (set! alist '(apples pears bananas))
(apples pears bananas)
festival> (car alist)
apples
festival> (cdr alist)
(pears bananas)
festival> (set! blist (cons 'oranges alist))
(oranges apples pears bananas)
festival> (append alist blist)
(apples pears bananas oranges apples pears bananas)
festival> (cons alist blist)
((apples pears bananas) oranges apples pears bananas)
festival> (length alist)
3
festival> (length (append alist blist))
7
```

8.3 Scheme Festival specifics

There a number of additions to SIOD that are Festival specific though still part of the Lisp system rather than the synthesis functions per se.

By convention if the first statement of a function is a string, it is treated as a documentation string. The string will be printed when help is requested for that function symbol.

In interactive mode if the function `:backtrace` is called (within parenthesis) the previous stack trace is displayed. Calling `:backtrace` with a numeric argument will display that particular stack frame in full. Note that any command other than `:backtrace` will reset the trace. You may optionally call

```
(set_backtrace t)
```

Which will cause a backtrace to be displayed whenever a Scheme error occurs. This can be put in your ``.festivalrc'` if you wish. This is especially useful when running Festival in non-interactive mode (batch or script mode) so that more information is printed when an error occurs.

A *hook* in Lisp terms is a position within some piece of code where a user may specify their own customization. The notion is used heavily in Emacs. In Festival there a number of places where hooks are used. A hook variable contains either a function or list of functions that are to be applied at some point in the processing. For example the `after_synth_hooks` are applied after synthesis has been applied to allow specific customization such as resampling or modification of the gain of the synthesized waveform. The Scheme function `apply_hooks` takes a hook variable as argument and an object and applies the function/list of functions in turn to the object.

When an error occurs in either Scheme or within the C++ part of Festival by default the system jumps to the top level, resets itself and continues. Note that errors are usually serious things, pointing to bugs in parameters or code. Every effort has been made to ensure that the processing of text never causes errors in Festival. However when using Festival as a development system it is often that errors occur in code.

Sometimes in writing Scheme code you know there is a potential for an error but you wish to ignore that and continue on to the next thing without exiting or stopping and returning to the top level. For example you are processing a number of utterances from a database and some files containing the descriptions have errors in them but you want your processing to continue through every utterance that can be processed rather than stopping 5 minutes after you gone home after setting a big batch job for overnight.

Festival's Scheme provides the function `unwind-protect` which allows the catching of errors and then continuing normally. For example suppose you have the function `process_utt` which takes a filename and does things which you know might cause an error. You can write the following to ensure you continue processing even in an error occurs.

```
(unwind-protect
  (process_utt filename)
  (begin
    (format t "Error found in processing %s\n" filename)
    (format t "continuing\n"))))
```

The `unwind-protect` function takes two arguments. The first is evaluated and if no error occurs the value returned from that expression is returned. If an error does occur while evaluating the first expression, the second expression is evaluated. `unwind-protect` may be used recursively. Note that all files opened while evaluating the first expression are closed if an error occurs. All global variables outside the scope of the `unwind-protect` will be left as they were set up until the error. Care should be taken in using this function but its power is necessary to be able to write robust Scheme code.

8.4 Scheme I/O

Different Scheme's may have quite different implementations of file i/o functions so in this section we will describe the basic functions in Festival SIOD regarding i/o.

Simple printing to the screen may be achieved with the function `print` which prints the given s-expression to the screen. The printed form is preceded by a new line. This is often useful for debugging but isn't really powerful enough for much else.

Files may be opened and closed and referred to file descriptors in a direct analogy to C's `stdio` library. The SIOD functions `fopen` and `fclose` work in the exactly the same way as their equivalently named partners in C.

The `format` command follows the command of the same name in Emacs and a number of other Lisps. C programmers can think of it as `fprintf`. `format` takes a file descriptor, format string and arguments to print. The file description may be a file descriptor as returned by the Scheme function `fopen`, it may also be `t` which means the output will be directed as standard out (cf. `printf`). A third possibility is `nil` which will cause the output to be printed to a string which is returned (cf. `sprintf`).

The format string closely follows the format strings in ANSI C, but it is not the same. Specifically the directives currently supported are, `%%`, `%d`, `%x`, `%s`, `%f`, `%g` and `%c`. All modifiers for these are also supported. In addition `%l` is provided for printing of Scheme objects as objects.

For example

```
(format t "%03d %3.4f %s %l %l %l\n" 23 23 "abc" "abc" '(a b d) utt1)
```

will produce

```
023 23.0000 abc "abc" (a b d) #<Utterance 32f228>
```

on standard output.

When large lisp expressions are printed they are difficult to read because of the parentheses. The function `pprintf` prints an expression to a file description (or `t` for standard out). It prints so the s-expression is nicely lined up and indented. This is often called pretty printing in Lisps.

For reading input from terminal or file, there is currently no equivalent to `scanf`. Items may only be read as Scheme expressions. The command

```
(load FILENAME t)
```

will load all s-expressions in `FILENAME` and return them, unevaluated as a list. Without the third argument the `load` function will load and evaluate each s-expression in the file.

To read individual s-expressions use `readfp`. For example

```
(let ((fd (fopen trainfile "r"))
      (entry)
      (count 0))
  (while (not (equal? (set! entry (readfp fd)) (eof-val)))
    (if (string-equal (car entry) "home")
        (set! count (+ 1 count))))
  (fclose fd))
```


To convert a symbol whose print name is a number to a number use `parse-number`. This is the equivalent to `atof` in C.

Note that, all i/o from Scheme input files is assumed to be basically some form of Scheme data (though can be just numbers, tokens). For more elaborate analysis of incoming data it is possible to use the text tokenization functions which offer a fully programmable method of reading data.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

9. TTS

Festival supports text to speech for raw text files. If you are not interested in using Festival in any other way except as black box for rendering text as speech, the following method is probably what you want.

```
festival --tts myfile
```

This will say the contents of ``myfile'`. Alternatively text may be submitted on standard input

```
echo hello world | festival --tts
cat myfile | festival --tts
```

Festival supports the notion of *text modes* where the text file type may be identified, allowing Festival to process the file in an appropriate way. Currently only two types are considered stable: `STML` and `raw`, but other types such as `email`, `HTML`, `Latex`, etc. are being developed and discussed below. This follows the idea of buffer modes in Emacs where a file's type can be utilized to best display the text. Text mode may also be selected based on a filename's extension.

Within the command interpreter the function `tts` is used to render files as text; it takes a filename and the text mode as arguments.

- [9.1 Utterance chunking](#) From text to utterances
- [9.2 Text modes](#) Mode specific text analysis
- [9.3 Example text mode](#) An example mode for reading email

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

9.1 Utterance chunking

Text to speech works by first tokenizing the file and chunking the tokens into utterances. The definition of utterance breaks is determined by the utterance tree in variable `eou_tree`. A default version is given in ``lib/tts.scm'`. This uses a decision tree to determine what signifies an utterance break. Obviously blank lines are probably the most reliable, followed by certain punctuation. The confusion of the use of periods for both sentence breaks and abbreviations requires some more heuristics to best guess their different use. The following tree is currently used which works better than simply using punctuation.

```

(defvar eou_tree
'((n.whitespace matches ".*\n.*\n\\(\\.\\|\\n\\)*") ;; 2 or more newlines
  ((1))
  ((punc in ("?" ":" "!"))
    ((1))
    ((punc is ".")
      ;; This is to distinguish abbreviations vs periods
      ;; These are heuristics
      ((name matches "\\(.*\\.?.*\\|[A-Z][A-Za-z]?[A-Za-z]?\\.\\|etc\\.\\)")
        ((n.whitespace is " ")
          ((0)) ;; if abbrev single space isn't enough for break
          ((n.name matches "[A-Z].*")
            ((1))
            ((0))))))
  ((n.whitespace is " ") ;; if it doesn't look like an abbreviation
    ((n.name matches "[A-Z].*") ;; single space and non-cap is no break
      ((1))
      ((0))
      ((1))))
  ((0))))

```

The token items this is applied to will always (except in the end of file case) include one following token, so look ahead is possible. The "n." and "p." and "p.p." prefixes allow access to the surrounding token context. The features name, whitespace and punc allow access to the contents of the token itself. At present there is no way to access the lexicon form this tree which unfortunately might be useful if certain abbreviations were identified as such there.

Note these are heuristics and written by hand not trained from data, though problems have been fixed as they have been observed in data. The above rules may make mistakes where abbreviations appear at end of lines, and when improper spacing and capitalization is used. This is probably worth changing, for modes where more casual text appears, such as email messages and USENET news messages. A possible improvement could be made by analysing a text to find out its basic threshold of utterance break (i.e. if no full stop, two spaces, followed by a capitalized word sequences appear and the text is of a reasonable length then look for other criteria for utterance breaks).

Ultimately what we are trying to do is to chunk the text into utterances that can be synthesized quickly and start to play them quickly to minimise the time someone has to wait for the first sound when starting synthesis. Thus it would be better if this chunking were done on *prosodic phrases* rather than chunks more similar to linguistic sentences. Prosodic phrases are bounded in size, while sentences are not.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

9.2 Text modes

We do not believe that all texts are of the same type. Often information about the general contents of file will aid synthesis greatly. For example in Latex files we do not want to here "left brace, backslash e m" before each emphasized word, nor do we want to necessarily hear formatting commands. Festival offers a basic method for specifying customization rules depending on the *mode* of the text. By type we are following the notion of modes in Emacs and eventually will allow customization at a similar level.

Modes are specified as the third argument to the function `tts`. When using the Emacs interface to Festival the buffer mode is automatically passed as the text mode. If the mode is not supported a warning message is printed and the raw text mode is used.

Our initial text mode implementation allows configuration both in C++ and in Scheme. Obviously in C++ almost anything can be done but it is not as easy to reconfigure without recompilation. Here we will discuss those modes which can be fully configured at run time.

A text mode may contain the following

filter

A Unix shell program filter that processes the text file in some appropriate way. For example for email it might remove uninteresting headers and just output the subject, from line and the message body. If not specified, an identity filter is used.

init_function

This (Scheme) function will be called before any processing will be done. It allows further set up of tokenization rules and voices etc.

exit_function

This (Scheme) function will be called at the end of any processing allowing resetting of tokenization rules etc.

analysis_mode

If analysis mode is `xml` the file is read through the built in XML parser `rxp`. Alternatively if analysis mode is `xxml` the filter should use an SGML normalising parser and the output is processed in a way suitable for it. Any other value is ignored.

These mode specific parameters are specified in the a-list held in `tts_text_modes`.

When using Festival in Emacs the emacs buffer mode is passed to Festival as the text mode.

Note that above mechanism is not really designed to be re-entrant, this should be addressed in later versions.

Following the use of auto-selection of mode in Emacs, Festival can auto-select the text mode based on the filename given when no explicit mode is given. The Lisp variable `auto-text-mode-alist` is a list of dotted pairs of regular expression and mode name. For example to specify that the `email` mode is to be used for files ending in ``.email'` we would add to the current `auto-text-mode-alist` as follows

```
(set! auto-text-mode-alist
      (cons (cons "\\`.email$" 'email)
            auto-text-mode-alist))
```

If the function `tts` is called with a mode other than `nil` that mode overrides any specified by the `auto-text-mode-alist`. The mode `fundamental` is the explicit "null" mode, it is used when no mode is specified in the function `tts`, and match is found in `auto-text-mode-alist` or the specified mode is not found.

By convention if a requested text model is not found in `tts_text_modes` the file ``.MODENAME-mode'` will be required. Therefore if you have the file ``.MODENAME-mode.scm'` in your library then it will be automatically loaded on reference. Modes may be quite large and it is not necessary have Festival load them all at start up time.

Because of the `auto-text-mode-alist` and the auto loading of currently undefined text modes you can use Festival like

```
festival --tts example.email
```

Festival will automatically synthesize ``.example.email'` in text mode `email`.

If you add your own personal text modes you should do the following. Suppose you've written an HTML mode. You have named it ``.html-mode.scm'` and put it in `~/home/awb/lib/festival/`. In your ``.festivalrc'` first identify your personal Festival library directory by adding it to `lib-path`.

```
(set! lib-path (cons "/home/awb/lib/festival/" lib-path))
```

Then add the definition to the `auto-text-mode-alist` that file names ending ``.html'` or ``.htm'` should be read in HTML mode.

```
(set! auto-text-mode-alist
      (cons (cons "\\`.html?$" 'html)
            auto-text-mode-alist))
```

Then you may synthesize an HTML file either from Scheme

```
(tts "example.html" nil)
```

Or from the shell command line

```
festival --tts example.html
```

Anyone familiar with modes in Emacs should recognise that the process of adding a new text mode to Festival is very similar to adding a new buffer mode to Emacs.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

9.3 Example text mode

Here is a short example of a tts mode for reading email messages. It is by no means complete but is a start at showing how you can customize tts modes without writing new C++ code.

The first task is to define a filter that will take a saved mail message and remove extraneous headers and just leave the from line, subject and body of the message. The filter program is given a file name as its first argument and should output the result on standard out. For our purposes we will do this as a shell script.

```
#!/bin/sh
# Email filter for Festival tts mode
# usage: email_filter mail_message >tidied_mail_message
grep "^From: " $1
echo
grep "^Subject: " $1
echo
# delete up to first blank line (i.e. the header)
sed '1,/^\$/ d' $1
```

Next we define the email init function, which will be called when we start this mode. What we will do is save the current token to words function and slot in our own new one. We can then restore the previous one when we exit.

```
(define (email_init_func)
  "Called on starting email text mode."
  (set! email_previous_t2w_func token_to_words)
  (set! english_token_to_words email_token_to_words)
  (set! token_to_words email_token_to_words))
```

Note that *both* `english_token_to_words` and `token_to_words` should be set to ensure that our new token to word function is still used when we change voices.

The corresponding end function puts the token to words function back.

```
(define (email_exit_func)
  "Called on exit email text mode."
  (set! english_token_to_words email_previous_t2w_func)
  (set! token_to_words email_previous_t2w_func))
```

Now we can define the email specific token to words function. In this example we deal with two specific cases. First we deal with the common form of email addresses so that the angle brackets are not pronounced. The second points are to recognise quoted text and immediately change the the speaker to the alternative speaker.

```
(define (email_token_to_words token name)
  "Email specific token to word rules."
  (cond
```

This first condition identifies the token as a bracketed email address and removes the brackets and splits the token

into name and IP address. Note that we recursively call the function `email_previous_t2w_func` on the email name and IP address so that they will be pronounced properly. Note that because that function returns a *list* of words we need to append them together.

```
((string-matches name "<.*.*>")
  (append
    (email_previous_t2w_func token
      (string-after (string-before name "@") "<"))
    (cons
      "at"
      (email_previous_t2w_func token
        (string-before (string-after name "@") ">"))))))
```

Our next condition deals with identifying a greater than sign being used as a quote marker. When we detect this we select the alternative speaker, even though it may already be selected. We then return no words so the quote marker is not spoken. The following condition finds greater than signs which are the first token on a line.

```
((and (string-matches name ">")
      (string-matches (item.feats token "whitespace")
        "[ \t\n]*\n *"))
  (voice_don_diphone)
  nil ;; return nothing to say
)
```

If it doesn't match any of these we can go ahead and use the builtin token to words function. Actually, we call the function that was set before we entered this mode to ensure any other specific rules still remain. But before that we need to check if we've had a newline with doesn't start with a greater than sign. In that case we switch back to the primary speaker.

```
(t ;; for all other cases
  (if (string-matches (item.feats token "whitespace")
    ".*\n[ \t\n]*")
    (voice_rab_diphone))
  (email_previous_t2w_func token name))))
```

In addition to these we have to actually declare the text mode. This we do by adding to any existing modes as follows.

```
(set! tts_text_modes
  (cons
    (list
      'email ;; mode name
      (list ;; email mode params
        (list 'init_func email_init_func)
        (list 'exit_func email_exit_func)
        '(filter "email_filter"))
      tts_text_modes))
```

This will now allow simple email messages to be dealt with in a mode specific way.

An example mail message is included in `examples/ex1.email`. To hear the result of the above text mode start Festival, load in the email mode descriptions, and call TTS on the example file.

```
(tts ".../examples/ex1.email" 'email)
```

The above is very short of a real email mode but does illustrate how one might go about building one. It should be reiterated that text modes are new in Festival and their most effective form has not been discovered yet. This will improve with time and experience.

10. XML/SGML mark-up

The ideas of a general, synthesizer system nonspecific, mark-up language for labelling text has been under discussion for some time. Festival has supported an SGML based markup language through multiple versions most recently STML (*sproat97*). This is based on the earlier SSML (Speech Synthesis Markup Language) which was supported by previous versions of Festival (*taylor96*). With this version of Festival we support *Sable* a similar mark-up language devised by a consortium from Bell Labs, Sub Microsystems, AT&T and Edinburgh, *sable98*. Unlike the previous versions which were SGML based, the implementation of Sable in Festival is now XML based. To the user they different is negligible but using XML makes processing of files easier and more standardized. Also Festival now includes an XML parser thus reducing the dependencies in processing Sable text.

Raw text has the problem that it cannot always easily be rendered as speech in the way the author wishes. Sable offers a well-defined way of marking up text so that the synthesizer may render it appropriately.

The definition of Sable is by no means settled and is still in development. In this release Festival offers people working on Sable and other XML (and SGML) based markup languages a chance to quickly experiment with prototypes by providing a DTD (document type descriptions) and the mapping of the elements in the DTD to Festival functions. Although we have not yet (personally) investigated facilities like cascading style sheets and generalized SGML specification languages like DSSSL we believe the facilities offer by Festival allow rapid prototyping of speech output markup languages.

Primarily we see Sable markup text as a language that will be generated by other programs, e.g. text generation systems, dialog managers etc. therefore a standard, easy to parse, format is required, even if it seems overly verbose for human writers.

For more information of Sable and access to the mailing list see

<http://www.cstr.ed.ac.uk/projects/sable.html>

10.1 Sable example	an example of Sable with descriptions
10.2 Supported Sable tags	Currently supported Sable tags
10.3 Adding Sable tags	Adding new Sable tags
10.4 XML/SGML requirements	Software environment requirements for use
10.5 Using Sable	Rendering Sable files as speech

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

10.1 Sable example

Here is a simple example of Sable marked up text

```

<?xml version="1.0"?>
<!DOCTYPE SABLE PUBLIC "-//SABLE//DTD SABLE speech mark up//EN"
    "Sable.v0_2.dtd"
[]>
<SABLE>
<SPEAKER NAME="male1">

The boy saw the girl in the park <BREAK/> with the telescope.
The boy saw the girl <BREAK/> in the park with the telescope.

Good morning <BREAK /> My name is Stuart, which is spelled
<RATE SPEED="-40%">
<SAYAS MODE="literal">stuart</SAYAS> </RATE>
though some people pronounce it
<PRON SUB="stoo art">stuart</PRON>. My telephone number
is <SAYAS MODE="literal">2787</SAYAS>.

I used to work in <PRON SUB="Buckloo">Buccleuch</PRON> Place,
but no one can pronounce that.

By the way, my telephone number is actually
<AUDIO SRC="http://www.cstr.ed.ac.uk/~awb/sounds/touchtone.2.au"/>
<AUDIO SRC="http://www.cstr.ed.ac.uk/~awb/sounds/touchtone.7.au"/>
<AUDIO SRC="http://www.cstr.ed.ac.uk/~awb/sounds/touchtone.8.au"/>
<AUDIO SRC="http://www.cstr.ed.ac.uk/~awb/sounds/touchtone.7.au"/>.
</SPEAKER>
</SABLE>

```

After the initial definition of the SABLE tags, through the file `Sable.v0_2.dtd`, which is distributed as part of Festival, the body is given. There are tags for identifying the language and the voice. Explicit boundary markers may be given in text. Also duration and intonation control can be explicitly specified as can new pronunciations of words. The last sentence specifies some external filenames to play at that point.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#)
[\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

10.2 Supported Sable tags

There is not yet a definitive set of tags but hopefully such a list will form over the next few months. As adding support for new tags is often trivial the problem lies much more in defining what tags there should be than in actually implementing them. The following are based on version 0.2 of Sable as described in http://www.cstr.ed.ac.uk/projects/sable_spec2.html, though some aspects are not currently supported in this implementation. Further updates will be announced through the Sable mailing list.

LANGUAGE

Allows the specification of the language through the ID attribute. Valid values in Festival are, `english`, `en1`, `spanish`, `en`, and others depending on your particular installation. For example

```
<LANGUAGE id="english"> ... </LANGUAGE>
```

If the language isn't supported by the particular installation of Festival "Some text in .." is said instead and the section is omitted.

SPEAKER

Select a voice. Accepts a parameter NAME which takes values `male1`, `male2`, `female1`, etc. There is currently no definition about what happens when a voice is selected which the synthesizer doesn't support. An example is

```
<SPEAKER name="male1"> ... </SPEAKER>
```

AUDIO

This allows the specification of an external waveform that is to be included. There are attributes for specifying volume and whether the waveform is to be played in the background of the following text or not. Festival as yet only supports insertion.

```
My telephone number is
<AUDIO SRC="http://www.cstr.ed.ac.uk/~awb/sounds/touchtone.2.au"/>
<AUDIO SRC="http://www.cstr.ed.ac.uk/~awb/sounds/touchtone.7.au"/>
<AUDIO SRC="http://www.cstr.ed.ac.uk/~awb/sounds/touchtone.8.au"/>
<AUDIO SRC="http://www.cstr.ed.ac.uk/~awb/sounds/touchtone.7.au"/>.
```

MARKER

This allows Festival to mark when a particular part of the text has been reached. At present the simply the value of the MARK attribute is printed. This is done some when that piece of text is analyzed, not when it is played. To use this in any real application would require changes to this tag's implementation.

Move the `<MARKER MARK="mouse" />` mouse to the top.

BREAK

Specifies a boundary at some LEVEL. Strength may be values Large, Medium, Small or a number. Note that this tag is an empty tag and must include the closing part within itself specification.

```
<BREAK LEVEL="LARGE" />
```

DIV

This signals an division. In Festival this causes an utterance break. A TYPE attribute may be specified but it is ignored by Festival.

PRON

Allows pronunciation of enclosed text to be explicitly given. It supports the attributes IPA for an IPA specification (not currently supported by Festival); SUB text to be substituted which can be in some form of phonetic spelling, and ORIGIN where the linguistic origin of the enclosed text may be identified to assist in etymologically sensitive letter to sound rules.

```
<PRON SUB="toe maa toe">tomato</PRON>
```

SAYAS

Allows identification of the enclosed tokens/text. The attribute MODE can take any of the following values: literal, date, time, phone, net, postal, currency, math, fraction, measure, ordinal, cardinal, or name. Further specification of type for dates (MDY, DMY etc) may be specified through the MODETYPE attribute.

```
As a test of marked-up numbers. Here we have
a year <SAYAS MODE="date">1998</SAYAS>,
an ordinal <SAYAS MODE="ordinal">1998</SAYAS>,
a cardinal <SAYAS MODE="cardinal">1998</SAYAS>,
a literal <SAYAS MODE="literal">1998</SAYAS>,
and phone number <SAYAS MODE="phone">1998</SAYAS>.
```

EMPH

To specify enclosed text should be emphasized, a LEVEL attribute may be specified but its value is currently ignored by Festival (besides the emphasis Festival generates isn't very good anyway).

```
The leaders of <EMPH>Denmark</EMPH> and <EMPH>India</EMPH> meet on
Friday.
```

PITCH

Allows the specification of pitch range, mid and base points.

```
Without his penguin, <PITCH BASE="-20%"> which he left at home, </PITCH>
he could not enter the restaurant.
```

RATE

Allows the specification of speaking rate

```
The address is <RATE SPEED="-40%"> 10 Main Street </RATE>.
```

VOLUME

Allows the specification of volume. Note in festival this causes an utterance break before and after this tag.

Please speak more `<VOLUME LEVEL="loud">loudly</VOLUME>`, except when I ask you to speak `<VOLUME LEVEL="quiet">in a quiet voice</VOLUME>`.

ENGINE

This allows specification of engine specific commands

An example is `<ENGINE ID="festival" DATA="our own festival speech synthesizer"> the festival speech synthesizer</ENGINE>` or the Bell Labs speech synthesizer.

These tags may change in name but they cover the aspects of speech mark up that we wish to express. Later additions and changes to these are expected.

See the files ``festival/examples/example.sable'` and ``festival/examples/example2.sable'` for working examples.

Note the definition of Sable is on going and there are likely to be later more complete implementations of sable for Festival as independent releases consult ``url://www.cstr.ed.ac.uk/projects/sable.html'` for the most recent updates.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

10.3 Adding Sable tags

We do not yet claim that there is a fixed standard for Sable tags but we wish to move towards such a standard. In the mean time we have made it easy in Festival to add support for new tags without, in general, having to change any of the core functions.

Two changes are necessary to add a new tags. First, change the definition in ``lib/Sable.v0_2.dtd'`, so that Sable files may use it. The second stage is to make Festival sensitive to that new tag. The example in `festival/lib/sable-mode.scm` shows how a new text mode may be implemented for an XML/SGML-based markup language. The basic point is that an identified function will be called on finding a start tag or end tags in the document. It is the tag-function's job to synthesize the given utterance if the tag signals an utterance boundary. The return value from the tag-function is the new status of the current utterance, which may remain unchanged or if the current utterance has been synthesized `nil` should be returned signalling a new utterance.

Note the hierarchical structure of the document is not available in this method of tag-functions. Any hierarchical state that must be preserved has to be done using explicit stacks in Scheme. This is an artifact due to the cross relationship to utterances and tags (utterances may end within start and end tags), and the desire to have all specification in Scheme rather than C++.

The tag-functions are defined in an elements list. They are identified with names such as `"(SABLE"` and `")SABLE"` denoting start and end tags respectively. Two arguments are passed to these tag functions, an assoc list of attributes and values as specified in the document and the current utterances. If the tag denotes an utterance break, call `xxml_synth` on `UTT` and return `nil`. If a tag (start or end) is found in the document and there is no corresponding tag-function it is ignored.

New features may be added to words with a start and end tag by adding features to the global `xxml_word_features`. Any features in that variable will be added to each word.

Note that this method may be used for both XML based languages and SGML based markup languages (though and external normalizing SGML parser is required in the SGML case). The type (XML vs SGML) is identified by the `analysis_type` parameter in the `tts` text mode specification.

10.4 XML/SGML requirements

Festival is distributed with `rxp` an XML parser developed by Richard Tobin of the Language Technology Group, University of Edinburgh. Sable is set up as an XML text mode so no further requirements or external programs are required to synthesize from Sable marked up text (unlike previous releases). Note that `rxp` is not a full validation parser and hence doesn't check some aspects of the file (tags within tags).

Festival still supports SGML based markup but in such cases requires an external SGML normalizing parser. We have tested ``nsgmls-1.0'` which is available as part of the SGML tools set ``sp-1.1.tar.gz'` which is available from <http://www.jclark.com/sp/index.html>. This seems portable between many platforms.

10.5 Using Sable

Support in Festival for Sable is as a text mode. In the command mode use the following to process an Sable file

```
(tts "file.sable" 'sable)
```

Also the automatic selection of mode based on file type has been set up such that files ending ``.sable'` will be automatically synthesized in this mode. Thus

```
festival --tts fred.sable
```

Will render `fred.sable'` as speech in Sable mode.

Another way of using Sable is through the Emacs interface. The `say-buffer` command will send the Emacs buffer mode to Festival as its `tts-mode`. If the Emacs mode is `stml` or `sgml` the file is treated as an sable file. See section [11. Emacs interface](#).

Many people experimenting with Sable (and TTS in general) often want all the waveform output to be saved to be played at a later date. The simplest way to do this is using the `text2wave` script. It respects the audio mode selection so

```
text2wave fred.sable -o fred.wav
```

Note this renders the file a single waveform (done by concatenating the waveforms for each utterance in the Sable file).

If you wish the waveform for each utterance in a file saved you can cause the `tts` process to save the waveforms during synthesis. A call to

```
festival> (save_waves_during_tts)
```

Any future call to `tts` will cause the waveforms to be saved in a file ``tts_file_xxx.wav'` where `xxx` is a number. A call to `(save_waves_during_tts_STOP)` will stop saving the waves. A message is printed when the waveform is saved otherwise people forget about this and wonder why their disk has filled up.

This is done by inserting a function in `tts_hooks` which saves the wave. To do other things to each utterances during TTS (such as saving the utterance structure), try redefining the function `save_tts_output` (see

festival/lib/tts.scm).

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

11. Emacs interface

One easy method of using Festival is via an Emacs interface that allows selection of text regions to be sent to Festival for rendering as speech.

``festival.el'` offers a new minor mode which offers an extra menu (in emacs-19 and 20) with options for saying a selected region, or a whole buffer, as well as various general control functions. To use this you must install ``festival.el'` in a directory where Emacs can find it, then add to your ``.emacs'` in your home directory the following lines.

```
(autoload 'say-minor-mode "festival" "Menu for using Festival." t)
(say-minor-mode t)
```

Successive calls to `say-minor-mode` will toggle the minor mode, switching the ``say'` menu on and off.

Note that the optional voice selection offered by the language sub-menu is not sensitive to actual voices supported by the your Festival installation. Hand customization is require in the ``festival.el'` file. Thus some voices may appear in your menu that your Festival doesn't support and some voices may be supported by your Festival that do not appear in the menu.

When the Emacs Lisp function `festival-say-buffer` or the menu equivalent is used the Emacs major mode is passed to Festival as the text mode.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

12. Phonemesets

The notion of phonemesets is important to a number of different subsystems within Festival. Festival supports multiple phonemesets simultaneously and allows mapping between sets when necessary. The lexicons, letter to sound rules, waveform synthesizers, etc. all require the definition of a phonemeset before they will operate.

A phonemeset is a set of symbols which may be further defined in terms of features, such as vowel/consonant, place of articulation for consonants, type of vowel etc. The set of features and their values must be defined with the phonemeset. The definition is used to ensure compatibility between sub-systems as well as allowing groups of phones in various prediction systems (e.g. duration)

A phonemeset definition has the form

```
(defPhoneSet
  NAME
  FEATUREDEFS
  PHONEDEFS )
```

The *NAME* is any unique symbol used e.g. `mrpa`, `darpa`, etc. *FEATUREDEFS* is a list of definitions each consisting of a feature name and its possible values. For example

```
(
  (vc + -)                ;; vowel consonant
  (vlength short long diphthong schwa 0) ;; vowel length
  ...
)
```

The third section is a list of phone definitions themselves. Each phone definition consists of a phone name and the values for each feature in the order the features were defined in the above section.

A typical example of a phoneset definition can be found in ``lib/mrpa_phones.scm'`.

Note the phoneset should also include a definition for any silence phones. In addition to the definition of the set the silence phone(s) themselves must also be identified to the system. This is done through the command `PhoneSet.silences`. In the `mrpa` set this is done by the command

```
(PhoneSet.silences '(#))
```

There may be more than one silence phone (e.g. breath, start silence etc.) in any phoneset definition. However the first phone in this set is treated special and should be canonical silence. Among other things, it is this phone that is inserted by the pause prediction module.

In addition to declaring phonesets, alternate sets may be selected by the command `PhoneSet.select`.

Phones in different sets may be automatically mapped between using their features. This mapping is not yet as general as it could be, but is useful when mapping between various phonesets of the same language. When a phone needs to be mapped from one set to another the phone with matching features is selected. This allows, at least to some extent, lexicons, waveform synthesizers, duration modules etc. to use different phonesets (though in general this is not advised).

A list of currently defined phonesets is returned by the function

```
(PhoneSet.list)
```

Note phonesets are often not defined until a voice is actually loaded so this list is not the list of of sets that are distributed but the list of sets that are used by currently loaded voices.

The name, phones, features and silences of the current phoneset may be accessed with the function

```
(PhoneSet.description nil)
```

If the argument to this function is a list, only those parts of the phoneset description named are returned. For example

```
(PhoneSet.description '(silences))
(PhoneSet.description '(silences phones))
```

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

13. Lexicons

A *Lexicon* in Festival is a subsystem that provides pronunciations for words. It can consist of three distinct parts: an addenda, typically short consisting of hand added words; a compiled lexicon, typically large (10,000s of words) which sits on disk somewhere; and a method for dealing with words not in either list.

13.2 Defining lexicons	Building new lexicons
13.3 Lookup process	Order of significance
13.4 Letter to sound rules	Dealing with unknown words
13.5 Building letter to sound rules	Building rules from data
13.6 Lexicon requirements	What should be in the lexicon
13.7 Available lexicons	Current available lexicons
13.8 Post-lexical rules	Modification of words in context

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

13.1 Lexical entries

Lexical entries consist of three basic parts, a head word, a part of speech and a pronunciation. The headword is what you might normally think of as a word e.g. `walk`, `chairs` etc. but it might be any token.

The part-of-speech field currently consist of a simple atom (or nil if none is specified). Of course there are many part of speech tag sets and whatever you mark in your lexicon must be compatible with the subsystems that use that information. You can optionally set a part of speech tag mapping for each lexicon. The value should be a reverse assoc-list of the following form

```
(lex.set.pos.map
  '((( punc fpunc) punc)
    (( nn nnp nns nnps ) n)))
```

All part of speech tags not appearing in the left hand side of a pos map are left unchanged.

The third field contains the actual pronunciation of the word. This is an arbitrary Lisp S-expression. In many of the lexicons distributed with Festival this entry has internal format, identifying syllable structure, stress markings and of course the phones themselves. In some of our other lexicons we simply list the phones with stress marking on each vowel.

Some typical example entries are

```
( "walkers" n ((( w oo ) 1) (( k @ z ) 0)) )
( "present" v ((( p r e ) 0) (( z @ n t ) 1)) )
( "monument" n ((( m o ) 1) (( n y u ) 0) (( m @ n t ) 0)) )
```

Note you may have two entries with the same headword, but different part of speech fields allow differentiation. For example

```
( "lives" n ((( l ai v z ) 1)) )
( "lives" v ((( l i v z ) 1)) )
```

See section [13.3 Lookup process](#), for a description of how multiple entries with the same headword are used during lookup.

By current conventions, single syllable function words should have no stress marking, while single syllable content words should be stressed.

NOTE: the POS field may change in future to contain more complex formats. The same lexicon mechanism (but different lexicon) is used for holding part of speech tag distributions for the POS prediction module.

13.2 Defining lexicons

As stated above, lexicons consist of three basic parts (compiled form, addenda and unknown word method) plus some other declarations.

Each lexicon in the system has a name which allows different lexicons to be selected from efficiently when switching between voices during synthesis. The basic steps involved in a lexicon definition are as follows.

First a new lexicon must be created with a new name

```
(lex.create "cstrlex")
```

A phone set must be declared for the lexicon, to allow both checks on the entries themselves and to allow phone mapping between different phone sets used in the system

```
(lex.set.phoneset "mrpa")
```

The phone set must be already declared in the system.

A compiled lexicon, the construction of which is described below, may be optionally specified

```
(lex.set.compile.file "/projects/festival/lib/dicts/cstrlex.out")
```

The method for dealing with unknown words, See section [13.4 Letter to sound rules](#), may be set

```
(lex.set.lts.method 'lts_rules)  
(lex.set.lts.ruleset 'nrl)
```

In this case we are specifying the use of a set of letter to sound rules originally developed by the U.S. Naval Research Laboratories. The default method is to give an error if a word is not found in the addenda or compiled lexicon. (This and other options are discussed more fully below.)

Finally addenda items may be added for words that are known to be common, but not in the lexicon and cannot reasonably be analysed by the letter to sound rules.

```
(lex.add.entry  
  '( "awb" n ((( ei ) 1) ((d uh) 1) ((b @ 1) 0) ((y uu) 0) ((b ii) 1))))  
(lex.add.entry  
  '( "cstr" n ((( s ii ) 1) (( e s ) 1) (( t ii ) 1) (( aa ) 1) ))  
(lex.add.entry  
  '( "Edinburgh" n ((( e m ) 1) (( b r @ ) 0))))
```

Using `lex.add.entry` again for the same word and part of speech will redefine the current pronunciation. Note these add entries to the *current* lexicon so its a good idea to explicitly select the lexicon before you add addenda entries, particularly if you are doing this in your own `.festivalrc` file.

For large lists, compiled lexicons are best. The function `lex.compile` takes two filename arguments, a file name containing a list of lexical entries and an output file where the compiled lexicon will be saved.

Compilation can take some time and may require lots of memory, as all entries are loaded in, checked and then sorted before being written out again. During compilation if some entry is malformed the reading process halts with a not so useful message. Note that if any of your entries include quote or double quotes the entries will probably be misparsed and cause such a weird error. In such cases try setting

```
(debug_output t)
```

before compilation. This will print out each entry as it is read in which should help to narrow down where the error is.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

13.3 Lookup process

When looking up a word, either through the C++ interface, or Lisp interface, a word is identified by its headword and part of speech. If no part of speech is specified, `nil` is assumed which matches any part of speech tag.

The lexicon look up process first checks the addenda, if there is a full match (head word plus part of speech) it is returned. If there is an addenda entry whose head word matches and whose part of speech is `nil` that entry is returned.

If no match is found in the addenda, the compiled lexicon, if present, is checked. Again a match is when both head word and part of speech tag match, or either the word being searched for has a part of speech `nil` or an entry has its tag as `nil`. Unlike the addenda, if no full head word and part of speech tag match is found, the first word in the lexicon whose head word matches is returned. The rationale is that the letter to sound rules (the next defence) are unlikely to be better than an given alternate pronunciation for a the word but different part of speech. Even more so given that as there is an entry with the head word but a different part of speech this word may have an unusual pronunciation that the letter to sound rules will have no chance in producing.

Finally if the word is not found in the compiled lexicon it is passed to whatever method is defined for unknown words. This is most likely a letter to sound module. See section [13.4 Letter to sound rules](#).

Optional pre- and post-lookup hooks can be specified for a lexicon. As a single (or list of) Lisp functions. The pre-hooks will be called with two arguments (word and features) and should return a pair (word and features). The post-hooks will be given a lexical entry and should return a lexical entry. The pre- and post-hooks do nothing by default.

Compiled lexicons may be created from lists of lexical entries. A compiled lexicon is *much* more efficient for look up than the addenda. Compiled lexicons use a binary search method while the addenda is searched linearly. Also it would take a prohibitively long time to load in a typical full lexicon as an addenda. If you have more than a few hundred entries in your addenda you should seriously consider adding them to your compiled lexicon.

Because many publicly available lexicons do not have syllable markings for entries the compilation method supports automatic syllabification. Thus for lexicon entries for compilation, two forms for the pronunciation field are supported: the standard full syllabified and stressed form and a simpler linear form found in at least the BEEP and CMU lexicons. If the pronunciation field is a flat atomic list it is assumed syllabification is required.

Syllabification is done by finding the minimum sonorant position between vowels. It is not guaranteed to be accurate but does give a solution that is sufficient for many purposes. A little work would probably improve this significantly. Of course syllabification requires the entry's phones to be in the current phone set. The sonorant values are calculated from the *vc*, *ctype*, and *cvox* features for the current phoneset. See ``src/arch/festival/Phone.cc:ph_sonority()'` for actual definition.

Additionally in this flat structure vowels (atoms starting with a, e, i, o or u) may have 1 2 or 0 appended marking stress. This is again following the form found in the BEEP and CMU lexicons.

Some example entries in the flat form (taken from BEEP) are

```
("table" nil (t eɪ l b l))
("suspicious" nil (s @ s p i l sh @ s))
```


Also if syllabification is required there is an opportunity to run a set of "letter-to-sound"-rules on the input (actually an arbitrary re-write rule system). If the variable `lex_lts_set` is set, the lts ruleset of that name is applied to the flat input before syllabification. This allows simple predictable changes such as conversion of final r into longer vowel for English RP from American labelled lexicons.

A list of all matching entries in the addenda and the compiled lexicon may be found by the function `lex.lookup_all`. This function takes a word and returns all matching entries irrespective of part of speech.

You can optionally intercept the words as they are looked up, and after they have been found through `pre_hooks` and `post_hooks` for each lexicon. This allows a function or list of functions to be applied to a word and feature before lookup or to the resulting entry after lookup. The following example shows how to add voice specific entries to a general lexicon without affecting other voices that use that lexicon.

For example suppose we were trying to use a Scottish English voice with the US English (cmu) lexicon. A number of entries will be inappropriate but we can redefine some entries thus

```
(set! cmu_us_awb::lexicon_addenda
 '(
  ("edinburgh" n ((eh d) 1) ((ax n) 0) ((b r ax) 0)))
  ("poem" n ((p ow) 1) ((y ax m) 0)))
  ("usual" n ((y uw) 1) ((zh ax l) 0)))
  ("air" n ((ey r) 1)))
  ("hair" n ((hh ey r) 1)))
  ("fair" n ((f ey r) 1)))
  ("chair" n ((ch ey r) 1))))
```

We can then define a function that checks to see if the word looked up is in the speaker specific exception list and use that entry instead.

```
(define (cmu_us_awb::cmu_lookup_post entry)
  "(cmu_us_awb::cmu_lookup_post entry)
  Speaker specific lexicon addeda."
  (let ((ne
        (assoc_string (car entry) cmu_us_awb::lexicon_addenda)))
    (if ne
        ne
        entry)))
```

And then for the particular voice set up we need to add both a selection part *and* a reset part. Thus following the FestVox conventions for voice set up.

```
(define (cmu_us_awb::select_lexicon)
  ...
  (lex.select "cmu")
  ;; Get old var for reset and to append our function to it
  (set! cmu_us_awb::old_cmu_post_hooks
    (lex.set.post_hooks nil))
  (lex.set.post_hooks
    (append cmu_us_awb::old_cmu_post_hooks
            (list cmu_us_awb::cmu_lookup_post)))
  ...
)

...

(define (cmu_us_awb::reset_lexicon)
  ...
  ;; reset CMU's post_hooks back to original
  (lex.set.post_hooks cmu_us_awb::old_cmu_post_hooks)
  ...)
```

)

The above isn't the most efficient way as the word is looked up first then it is checked with the speaker specific list.

The `pre_hooks` function are called with two arguments, the word and features, they should return a pair of word and features.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

13.4 Letter to sound rules

Each lexicon may define what action to take when a word cannot be found in the addenda or the compiled lexicon. There are a number of options which will hopefully be added to as more general letter to sound rule systems are added.

The method is set by the command

```
(lex.set.lts.method METHOD)
```

Where *METHOD* can be any of the following

```
`Error'  
  Throw an error when an unknown word is found (default).  
`lts_rules'  
  Use externally specified set of letter to sound rules (described below). The name of the rule set to use is defined with the lex.lts.ruleset function. This method runs one set of rules on an exploded form of the word and assumes the rules return a list of phonemes (in the appropriate set). If multiple instances of rules are required use the function method described next.  
`none'  
  This returns an entry with a nil pronunciation field. This will only be valid in very special circumstances.  
`FUNCTIONNAME'  
  Call this as a LISP function function name. This function is given two arguments: the word and the part of speech. It should return a valid lexical entry.
```

The basic letter to sound rule system is very simple but is powerful enough to build reasonably complex letter to sound rules. Although we've found trained LTS rules better than hand written ones (for complex languages) where no data is available and rules must be hand written the following rule formalism is much easier to use than that generated by the LTS training system (described in the next section).

The basic form of a rule is as follows

```
( LEFTCONTEXT [ ITEMS ] RIGHTCONTEXT = NEWITEMS )
```

This interpretation is that if *ITEMS* appear in the specified right and left context then the output string is to contain *NEWITEMS*. Any of *LEFTCONTEXT*, *RIGHTCONTEXT* or *NEWITEMS* may be empty. Note that *NEWITEMS* is written to a different "tape" and hence cannot feed further rules (within this ruleset). An example is

```
( # [ c h ] C = k )
```

The special character # denotes a word boundary, and the symbol C denotes the set of all consonants, sets are declared before rules. This rule states that a *ch* at the start of a word followed by a consonant is to be rendered as the *k* phoneme. Symbols in contexts may be followed by the symbol * for zero or more occurrences, or + for one or more occurrences.

The symbols in the rules are treated as set names if they are declared as such or as symbols in the input/output alphabets. The symbols may be more than one character long and the names are case sensitive.

The rules are tried in order until one matches the first (or more) symbol of the tape. The rule is applied adding the right hand side to the output tape. The rules are again applied from the start of the list of rules.

The function used to apply a set of rules if given an atom will explode it into a list of single characters, while if given a list will use it as is. This reflects the common usage of wishing to re-write the individual letters in a word to phonemes but without excluding the possibility of using the system for more complex manipulations, such as multi-pass LTS systems and phoneme conversion.

From lisp there are three basic access functions, there are corresponding functions in the C/C++ domain.

```
(lts.ruleset NAME SETS RULES)
```

Define a new set of lts rules. Where NAME is the name for this rule, SETS is a list of set definitions of the form (SETNAME e0 e1 . . .) and RULES are a list of rules as described above.

```
(lts.apply WORD RULESETNAME)
```

Apply the set of rules named RULESETNAME to WORD. If WORD is a symbol it is exploded into a list of the individual characters in its print name. If WORD is a list it is used as is. If the rules cannot be successfully applied an error is given. The result of (successful) application is returned in a list.

```
(lts.check_alpha WORD RULESETNAME)
```

The symbols in WORD are checked against the input alphabet of the rules named RULESETNAME. If they are all contained in that alphabet t is returned, else nil. Note this does not necessarily mean the rules will successfully apply (contexts may restrict the application of the rules), but it allows general checking like numerals, punctuation etc, allowing application of appropriate rule sets.

The letter to sound rule system may be used directly from Lisp and can easily be used to do relatively complex operations for analyzing words without requiring modification of the C/C++ system. For example the Welsh letter to sound rule system consists of three rule sets, first to explicitly identify epenthesis, then identify stressed vowels, and finally rewrite this augmented letter string to phonemes. This is achieved by the following function

```
(define (welsh_lts word features)
  (let (epen str wel)
    (set! epen (lts.apply (downcase word) 'newepen))
    (set! str (lts.apply epen 'newwelstr))
    (set! wel (lts.apply str 'newwel))
    (list word
          nil
          (lex.syllabify.phstress wel))))
```

The LTS method for the Welsh lexicon is set to welsh_lts, so this function is called when a word is not found in the lexicon. The above function first downcases the word and then applies the rulesets in turn, finally calling the syllabification process and returns a constructed lexically entry.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

13.5 Building letter to sound rules

As writing letter to sound rules by hand is hard and very time consuming, an alternative method is also available where a letter to sound system may be built from a lexicon of the language. This technique has successfully been used from English (British and American), French and German. The difficulty and appropriateness of using letter to sound rules is very language dependent,

The following outlines the processes involved in building a letter to sound model for a language given a large lexicon of pronunciations. This technique is likely to work for most European languages (including Russian) but doesn't seem particularly suitable for very language alphabet languages like Japanese and Chinese. The process described here is not (yet) fully automatic but the hand intervention required is small and may easily be done even by people with only a very little knowledge of the language being dealt with.

The process involves the following steps

- Pre-processing lexicon into suitable training set
- Defining the set of allowable pairing of letters to phones. (We intend to do this fully automatically in future versions).
- Constructing the probabilities of each letter/phone pair.
- Aligning letters to an equal set of phones/_epsilon_.
- Extracting the data by letter suitable for training.
- Building CART models for predicting phone from letters (and context).
- Building additional lexical stress assignment model (if necessary).

All except the first two stages of this are fully automatic.

Before building a model its wise to think a little about what you want it to do. Ideally the model is an auxiliary to the lexicon so only words not found in the lexicon will require use of the letter to sound rules. Thus only unusual forms are likely to require the rules. More precisely the most common words, often having the most non-standard pronunciations, should probably be explicitly listed always. It is possible to reduce the size of the lexicon (sometimes drastically) by removing all entries that the training LTS model correctly predicts.

Before starting it is wise to consider removing some entries from the lexicon before training, I typically will remove words under 4 letters and if part of speech information is available I remove all function words, ideally only training from nouns verbs and adjectives as these are the most likely forms to be unknown in text. It is useful to have morphologically inflected and derived forms in the training set as it is often such variant forms that not found in the lexicon even though their root morpheme is. Note that in many forms of text, proper names are the most common form of unknown word and even the technique presented here may not adequately cater for that form of unknown words (especially if they unknown words are non-native names). This is all stating that this may or may not be appropriate for your task but the rules generated by this learning process have in the examples we've done been much better than what we could produce by hand writing rules of the form described in the previous section.

First preprocess the lexicon into a file of lexical entries to be used for training, removing functions words and changing the head words to all lower case (may be language dependent). The entries should be of the form used for input for Festival's lexicon compilation. Specifical the pronunciations should be simple lists of phones (no syllabification). Depending on the language, you may wish to remve the stressing--for examples here we have though later tests suggest that we should keep it in even for English. Thus the training set should look something like

```
("table" nil (t eɪ b l))
("suspicious" nil (s @ s p i sh @ s))
```

It is best to split the data into a training set and a test set if you wish to know how well your training has worked. In our tests we remove every tenth entry and put it in a test set. Note this will mean our test results are probably better than if we removed say the last ten in every hundred.

The second stage is to define the set of allowable letter to phone mappings irrespective of context. This can sometimes be initially done by hand then checked against the training set. Initially construct a file of the form

```
(require 'lts_build)
(set! allowables
  '((a _epsilon_)
    (b _epsilon_)
    (c _epsilon_)
    ...
    (y _epsilon_)
    (z _epsilon_)
    (# #)))
```

All letters that appear in the alphabet should (at least) map to _epsilon_, including any accented characters that appear in that language. Note the last two hashes. These are used by to denote beginning and end of word and are automatically added during training, they must appear in the list and should only map to themselves.

To incrementally add to this allowable list run festival as

```
festival allowables.scm
```

and at the prompt type

```
festival> (cumulate-pairs "oald.train")
```

with your train file. This will print out each lexical entry that couldn't be aligned with the current set of allowables. At the start this will be every entry. Looking at these entries add to the allowables to make alignment work. For example if the following word fails

```
("abate" nil (ah b ey t))
```

Add ah to the allowables for letter a, b to b, ey to a and t to letter t. After doing that restart festival and call cumulate-pairs again. Incrementally add to the allowable pairs until the number of failures becomes acceptable. Often there are entries for which there is no real relationship between the letters and the pronunciation such as in abbreviations and foreign words (e.g. "aaa" as "t r ih p ax l ey"). For the lexicons I've used the technique on less than 10 per thousand fail in this way.

It is worth while being consistent on defining your set of allowables. (At least) two mappings are possible for the letter sequence ch--having letter c go to phone ch and letter h go to `_epsilon_` and also letter c go to phone `_epsilon_` and letter h goes to ch. However only one should be allowed, we preferred c to ch.

It may also be the case that some letters give rise to more than one phone. For example the letter x in English is often pronounced as the phone combination k and s. To allow this, use the multiphone k-s. Thus the multiphone k-s will be predicted for x in some context and the model will separate it into two phones while it also ignoring any predicted `_epsilon_`. Note that multiphone units are relatively rare but do occur. In English, letter x give rise to a few, k-s in taxi, g-s in example, and sometimes g-zh and k-sh in luxury. Others are w-ah in one, t-s in pizza, y-uw in new (British), ah-m in -ism etc. Three phone multiphone are much rarer but may exist, they are not supported by this code as is, but such entries should probably be ignored. Note the - sign in the multiphone examples is significant and is used to identify multiphones.

The allowables for OALD end up being

```
(set! allowables
  (
    ((a _epsilon_ ei aa a e@ @ oo au o i ou ai uh e)
     (b _epsilon_ b )
     (c _epsilon_ k s ch sh @-k s t-s)
     (d _epsilon_ d dh t jh)
     (e _epsilon_ @ ii e e@ i @@ i@ uu y-uu ou ei aa oi y y-u@ o)
     (f _epsilon_ f v )
     (g _epsilon_ g jh zh th f ng k t)
     (h _epsilon_ h @ )
     (i _epsilon_ i@ i @ ii ai @@ y ai-@ aa a)
     (j _epsilon_ h zh jh i y )
     (k _epsilon_ k ch )
     (l _epsilon_ l @-l l-l)
     (m _epsilon_ m @-m n)
     (n _epsilon_ n ng n-y )
     (o _epsilon_ @ ou o oo uu u au oi i @@ e uh w u@ w-uh y-@)
     (p _epsilon_ f p v )
     (q _epsilon_ k )
     (r _epsilon_ r @@ @-r)
     (s _epsilon_ z s sh zh )
     (t _epsilon_ t th sh dh ch d )
     (u _epsilon_ uu @ w @@ u uh y-uu u@ y-u@ y-u i y-uh y-@ e)
     (v _epsilon_ v f )
     (w _epsilon_ w uu v f u)
     (x _epsilon_ k-s g-z sh z k-sh z g-zh )
```

```
(y _epsilon_ i ii i@ ai uh y @ ai-@)
(z _epsilon_ z t-s s zh )
(# #)
))
```

Note this is an exhaustive list and (deliberately) says nothing about the contexts or frequency that these letter to phone pairs appear. That information will be generated automatically from the training set.

Once the number of failed matches is significantly low enough let `cummulate-pairs` run to completion. This counts the number of times each letter/phone pair occurs in allowable alignments.

Next call

```
festival> (save-table "oald-")
```

with the name of your lexicon. This changes the cummulation table into probabilities and saves it.

Restart festival loading this new table

```
festival allowables.scm oald-pl-table.scm
```

Now each word can be aligned to an equally-lengthed string of phones, epsilon and multiphones.

```
festival> (aligndata "oald.train" "oald.train.align")
```

Do this also for you test set.

This will produce entries like

```
aaronson _epsilon_ aa r ah n s ah n
abandon ah b ae n d ah n
abate ah b ey t _epsilon_
abbe ae b _epsilon_ iy
```

The next stage is to build features suitable for `wagon' to build models. This is done by

```
festival> (build-feat-file "oald.train.align" "oald.train.feats")
```

Again the same for the test set.

Now you need to constructure a description file for `wagon' for the given data. The can be done using the script `make_wgn_desc' provided with the speech tools

Here is an example script for building the models, you will need to modify it for your particualr database but it shows the basic processes

```

for i in a b c d e f g h i j k l m n o p q r s t u v w x y z
do
  # Stop value for wagon
  STOP=2
  echo letter $i STOP $STOP
  # Find training set for letter $i
  cat oald.train.feats |
  awk '{if ($6 == "'$i'") print $0}' >ltsdataTRAIN.$i.feats
  # split training set to get heldout data for stepwise testing
  traintest ltsdataTRAIN.$i.feats
  # Extract test data for letter $i
  cat oald.test.feats |
  awk '{if ($6 == "'$i'") print $0}' >ltsdataTEST.$i.feats
  # run wagon to predict model
  wagon -data ltsdataTRAIN.$i.feats.train -test ltsdataTRAIN.$i.feats.test \
        -stepwise -desc ltsOALD.desc -stop $STOP -output lts.$i.tree
  # Test the resulting tree against
  wagon_test -heap 2000000 -data ltsdataTEST.$i.feats -desc ltsOALD.desc \
            -tree lts.$i.tree
done

```

The script `traintest` splits the given file `X` into `X.train` and `X.test` with every tenth line in `X.test` and the rest in `X.train`.

This script can take a significant amount of time to run, about 6 hours on a Sun Ultra 140.

Once the models are created they must be collected together into a single list structure. The trees generated by `wagon` contain fully probability distributions at each leaf, at this time this information can be removed as only the most probable will actually be predicted. This substantially reduces the size of the trees.

```
(merge_models 'oald_lts_rules "oald_lts_rules.scm")
```

(merge_models is defined within `lts_build.scm`) The given file will contain a set! for the given variable name to an assoc list of letter to trained tree. Note the above function naively assumes that the letters in the alphabet are the 26 lower case letters of the English alphabet, you will need to edit this adding accented letters if required. Note that adding "" (single quote) as a letter is a little tricky in scheme but can be done--the command (intern " ' ") will give you the symbol for single quote.

To test a set of lts models load the saved model and call the following function with the test align file

```

festival oald-table.scm oald_lts_rules.scm
festival> (lts_testset "oald.test.align" oald_lts_rules)

```

The result (after showing all the failed ones), will be a table showing the results for each letter, for all letters and for complete words. The failed entries may give some notion of how good or bad the result is, sometimes it will be simple vowel differences, long versus short, schwa versus full vowel, other times it may be who consonants missing. Remember the ultimate quality of the letter sound rules is how adequate they are at providing *acceptable* pronunciations rather than how good the numeric score is.

For some languages (e.g. English) it is necessary to also find a tree pattern for unknown words. Ultimately for this to work well you need to know the morphological decomposition of the word. At present we provide a CART trained system to predict stress patterns for English. It does get 94.6% correct for an unseen test set but that isn't really very good. Later tests suggest that predicting stressed and unstressed phones directly is actually better for getting whole words correct even though the models do slightly worse on a per phone basis *black98*.

As the lexicon may be a large part of the system we have also experimented with removing entries from the lexicon if the letter to sound rules system (and tree assignment system) can't correctly predict them. For OALD this allows us to half the size of the lexicon, it could possibly allow more if a certain amount of fuzzy acceptance was allowed (e.g. with schwa). For other languages the gain here can be very significant, for German and French we can reduce the

lexicon by over 90%. The function `reduce_lexicon` in ``festival/lib/lts_build.scm'` was used to do this. A discussion of using the above technique as a dictionary compression method is discussed in *page198*. A morphological decomposition algorithm, like that described in *black91*, may even help more.

The technique described in this section and its relative merits with respect to a number of languages/lexicons and tasks is discussed more fully in *black98*.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

13.6 Lexicon requirements

For English there are a number of assumptions made about the lexicon which are worthy of explicit mention. If you are basically going to use the existing token rules you should try to include at least the following in any lexicon that is to work with them.

- The letters of the alphabet, when a token is identified as an acronym it is spelled out. The tokenization assumes that the individual letters of the alphabet are in the lexicon with their pronunciations. They should be identified as nouns. (This is to distinguish a as a determiner which can be schwa'd from a as a letter which cannot.) The part of speech should be `nn` by default, but the value of the variable `token.letter_pos` is used and may be changed if this is not what is required.
- One character symbols such as dollar, at-sign, percent etc. Its difficult to get a complete list and to know what the pronunciation of some of these are (e.g hash or pound sign). But the letter to sound rules cannot deal with them so they need to be explicitly listed. See the list in the function `mrpa_addend` in ``festival/lib/dicts/oald/oaldlex.scm'`. This list should also contain the control characters and eight bit characters.
- The possessive 's should be in your lexicon as schwa and voiced fricative (z). It should be in twice, once as part speech type `pos` and once as `n` (used in plurals of numbers acronyms etc. e.g 1950's). 's is treated as a word and is separated from the tokens it appears with. The post-lexical rule (the function `postlex_apos_s_check`) will delete the schwa and devoice the z in appropriate contexts. Note this post-lexical rule brazenly assumes that the unvoiced fricative in the phoneset is `s`. If it is not in your phoneset copy the function (it is in ``festival/lib/postlex.scm'`) and change it for your phoneset and use your version as a post-lexical rule.
- Numbers as digits (e.g. "1", "2", "34", etc.) should normally *not* be in the lexicon. The number conversion routines convert numbers to words (i.e. "one", "two", "thirty four", etc.).
- The word "unknown" or whatever is in the variable `token.unknown_word_name`. This is used in a few obscure cases when there just isn't anything that can be said (e.g. single characters which aren't in the lexicon). Some people have suggested it should be possible to make this a sound rather than a word. I agree, but Festival doesn't support that yet.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

13.7 Available lexicons

Currently Festival supports a number of different lexicons. They are all defined in the file ``lib/lexicons.scm'` each with a number of common extra words added to their addendas. They are

``CUVOALD'`

The Computer Users Version of Oxford Advanced Learner's Dictionary is available from the Oxford Text Archive [ftp://ota.ox.ac.uk/pub/ota/public/dicts/710](http://ota.ox.ac.uk/pub/ota/public/dicts/710). It contains about 70,000 entries and is a part of the BEEP lexicon. It is more consistent in its marking of stress though its syllable marking is not what works best for our synthesis methods. Many syllabic ``l's`, ``n's`, and ``m's`, mess up the syllabification algorithm, making results sometimes appear over reduced. It is however our current default lexicon. It is also the only lexicon with part of speech tags that can be distributed (for non-commercial use).

``CMU'`

This is automatically constructed from ``cmu_dict-0.4'` available from many places on the net (see

comp . speech archives). It is not in the mrpa phone set because it is American English pronunciation. Although mappings exist between its phoneset (`darpa`) and `mrpa` the results for British English speakers are not very good. However this is probably the biggest, most carefully specified lexicon available. It contains just under 100,000 entries. Our distribution has been modified to include part of speech tags on words we know to be homographs.

`mrpa`

A version of the CSTR lexicon which has been floating about for years. It contains about 25,000 entries. A new updated free version of this is due to be released soon.

`BEEP`

A British English rival for the `cmu_lex`. BEEP has been made available by Tony Robinson at Cambridge and is available in many archives. It contains 163,000 entries and has been converted to the `mrpa` phoneset (which was a trivial mapping). Although large, it suffers from a certain randomness in its stress markings, making use of it for synthesis dubious.

All of the above lexicons have some distribution restrictions (though mostly pretty light), but as they are mostly freely available we provide programs that can convert the originals into Festival's format.

The MOBY lexicon has recently been released into the public domain and will be converted into our format soon.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

13.8 Post-lexical rules

It is the lexicon's job to produce a pronunciation of a given word. However in most languages the most natural pronunciation of a word cannot be found in isolation from the context in which it is to be spoken. This includes such phenomena as reduction, phrase final devoicing and r-insertion. In Festival this is done by post-lexical rules.

PostLex is a module which is run after accent assignment but before duration and F0 generation. This is because knowledge of accent position is necessary for vowel reduction and other post lexical phenomena and changing the segmental items will affect durations.

The PostLex first applies a set of built in rules (which could be done in Scheme but for historical reasons are still in C++). It then applies the functions set in the hook `postlex_rules_hook`. These should be a set of functions that take an utterance and apply appropriate rules. This should be set up on a per voice basis.

Although a rule system could be devised for post-lexical sound rules it is unclear what the scope of them should be, so we have left it completely open. Our vowel reduction model uses a CART decision tree to predict which syllables should be reduced, while the "s" rule is very simple (shown in ``festival/lib/postlex.scm``).

The 's in English may be pronounced in a number of different ways depending on the preceding context. If the preceding consonant is a fricative or affricative and not a palatal labio-dental or dental a schwa is required (e.g. bench 's) otherwise no schwa is required (e.g. John 's). Also if the previous phoneme is unvoiced the "s" is rendered as an "s" while in all other cases it is rendered as a "z".

For our English voices we have a lexical entry for "s" as a schwa followed by a "z". We use a post lexical rule function called `postlex_apos_s_check` to modify the basic given form when required. After lexical lookup the segment relation contains the concatenation of segments directly from lookup in the lexicon. Post lexical rules are applied after that.

In the following rule we check each segment to see if it is part of a word labelled "s", if so we check to see if are we currently looking at the schwa or the z part, and test if modification is required

```

(define (postlex_apos_s_check utt)
  "(postlex_apos_s_check UTT)
  Deal with possessive s for English (American and British). Delete
  schwa of 's if previous is not a fricative or affricative, and
  change voiced to unvoiced s if previous is not voiced."
  (mapcar
   (lambda (seg)
     (if (string-equal "'s" (item.feats
                          seg "R:SylStructure.parent.parent.name"))
         (if (string-equal "a" (item.feats seg 'ph_vlmg))
             (if (and (member_string (item.feats seg 'p.ph_ctype)
                                     '(f a))
                     (not (member_string
                           (item.feats seg "p.ph_cplace")
                           '(d b g))))
                 t;; don't delete schwa
             (item.delete seg))
         (if (string-equal "-" (item.feats seg "p.ph_cvox"))
             (item.set_name seg "s")))))
         utt.relation.items utt 'Segment))
  utt)

```

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

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	internal structure of utterances
14.2 Utterance types	Type defined synthesis actions
14.3 Example utterance types	Some example utterances
14.4 Utterance modules	
14.5 Accessing an utterance	getting the data from the structure
14.6 Features	Features and features names
14.7 Utterance I/O	Saving and loading utterances

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

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 contain 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 conversion are, in general, as follows

Tokenization

Converting the string of characters into a list of tokens. Typically this means whitespace separated tokens 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 appropriate 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.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

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 outputting 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.)

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

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`.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

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`.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

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.feats 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.feats ITEM 'name).  
` (item.set_name ITEM NEWNAME) '  
  Sets name on ITEM to be NEWNAME. This is equivalent to (item.set_feats ITEM 'name NEWNAME)  
` (item.set_feats 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 leaf 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 dereferenced 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 dereferenced 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 value pairs. If ITEM is nil or unspecified an new empty item is added. If ITEM is already in this relation it is dereferenced from its current position (and an empty 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 dereferenced. 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 punctuation terminal nodes in the Syntax relation the call would be something like


```
(define (syntax_relation_punc p)
  (if (string-equal "punc" (item.feats (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 descendants or vice versa nil is returned 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.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

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 traversal 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:<relationname>.` operator. For example given an Item in the syllable relation `R:SyllStructure.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.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

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 FO file. This function loads, synthesizes and plays an utterance synthesized from these files. The loading is provided by the underlying function `utt.load.segf0`.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

15. Text analysis

[15.1 Tokenizing](#) Splitting text into tokens

[15.2 Token to word rules](#)

[15.3 Homograph disambiguation](#) "Wed 5 may wind US Sen up"

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

15.1 Tokenizing

A crucial stage in text processing is the initial tokenization of text. A *token* in Festival is an atom separated with whitespace from a text file (or string). If punctuation for the current language is defined, characters matching that punctuation are removed from the beginning and end of a token and held as features of the token. The default list of characters to be treated as white space is defined as

```
(defvar token.whitespace " \t\n\r")
```

While the default set of punctuation characters is

```
(defvar token.punctuation "\"'`. , ; ! ? ( ) { } [ ]")  
(defvar token.prepunctuation "\"'` ( [ ")
```

These are declared in ``lib/token.scm'` but may be changed for different languages, text modes etc.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

15.2 Token to word rules

Tokens are further analysed into lists of words. A word is an atom that can be given a pronunciation by the lexicon (or letter to sound rules). A token may give rise to a number of words or none at all.

For example the basic tokens

This pocket-watch was made in 1983.

would give a word relation of

this pocket watch was made in nineteen eighty three

Because the relationship between tokens and word in some cases is complex, a user function may be specified for translating tokens into words. This is designed to deal with things like numbers, email addresses, and other non-obvious pronunciations of tokens as zero or more words. Currently a builtin function `builtin_english_token_to_words` offers much of the necessary functionality for English but a user may further customize this.

If the user defines a function `token_to_words` which takes two arguments: a token item and a token name, it will be called by the `Token_English` and `Token_Any` modules. A substantial example is given as `english_token_to_words` in ``festival/lib/token.scm'`.

An example of this function is in ``lib/token.scm'`. It is quite elaborate and covers most of the common multi-word tokens in English including, numbers, money symbols, Roman numerals, dates, times, plurals of symbols, number ranges, telephone number and various other symbols.

Let us look at the treatment of one particular phenomena which shows the use of these rules. Consider the expression "\$12 million" which should be rendered as the words "twelve million dollars". Note the word "dollars" which is introduced by the "\$" sign, ends up after the end of the expression. There are two cases we need to deal with as there are two tokens. The first condition in the `cond` checks if the current token name is a money symbol, while the second condition check that following word is a magnitude (million, billion, trillion, zillion etc.) If that is the case the "\$" is removed and the remaining numbers are pronounced, by calling the builtin token to word function. The second condition deals with the second token. It confirms the previous is a money value (the same regular expression as before) and then returns the word followed by the word "dollars". If it is neither of these forms then the builtin function is called.

```
(define (token_to_words token name)
  "(token_to_words TOKEN NAME)
  Returns a list of words for NAME from TOKEN."
  (cond
    ((and (string-matches name "\\$[0-9,]+\\(\\. [0-9]+\\)?" )
          (string-matches (item.feats token "n.name") ".*illion.?"))
      (builtin_english_token_to_words token (string-after name "$")))
    ((and (string-matches (item.feats token "p.name")
                          "\\$[0-9,]+\\(\\. [0-9]+\\)?" )
          (string-matches name ".*illion.?"))
      (list
        name
        "dollars"))
    (t
     (builtin_english_token_to_words token name))))
```

It is valid to make some conditions return no words, though some care should be taken with that, as punctuation information may no longer be available to later processing if there are no words related to a token.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

15.3 Homograph disambiguation

Not all tokens can be rendered as words easily. Their context may affect the way they are to be pronounced. For example in the utterance

On May 5 1985, 1985 people moved to Livingston.

the tokens "1985" should be pronounced differently, the first as a year, "nineteen eighty five" while the second as a quantity "one thousand nine hundred and eighty five". Numbers may also be pronounced as ordinals as in the "5" above, it should be "fifth" rather than "five".

Also, the pronunciation of certain words cannot simply be found from their orthographic form alone. Linguistic part of speech tags help to disambiguate a large class of homographs, e.g. "lives". A part of speech tagger is included in Festival and discussed in [16. POS tagging](#). But even part of speech isn't sufficient in a number of cases. Words such as "bass", "wind", "bow" etc cannot be distinguished by part of speech alone, some semantic information is also required. As full semantic analysis of text is outwith the realms of Festival's capabilities some other method for disambiguation is required.

Following the work of *yarowsky96* we have included a method for identified tokens to be further labelled with extra tags to help identify their type. Yarowsky uses *decision lists* to identify different types for homographs. Decision lists are a restricted form of decision trees which have some advantages over full trees, they are easier to build and Yarowsky has shown them to be adequate for typical homograph resolution.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

15.3.1 Using disambiguators

Festival offers a method for assigning a `token_pos` feature to each token. It does so using Yarowsky-type disambiguation techniques. A list of disambiguators can be provided in the variable `token_pos_cart_trees`. Each disambiguator consists of a regular expression and a CART tree (which may be a decision list as they have the same format). If a token matches the regular expression the CART tree is applied to the token and the resulting class is assigned to the token via the feature `token_pos`. This is done by the `Token_POS` module.

For example, the follow disambiguator distinguishes "St" (street and saint) and "Dr" (doctor and drive).

```
"\\([dD][Rr]\\| [Ss][tT]\\)"
((n.name is 0)
  ((p.cap is 1)
    ((street))
    ((p.name matches "[0-9]*\\(1[sS][tT]\\|2[nN][dD]\\|3[rR][dD]\\|[0-9][tT][hH]\\)" )
      ((street))
      ((title))))
  ((punc matches ".*,.*")
    ((street))
    ((p.punc matches ".*,.*")
      ((title))
      ((n.cap is 0)
        ((street))
        ((p.cap is 0)
          ((p.name matches "[0-9]*\\(1[sS][tT]\\|2[nN][dD]\\|3[rR][dD]\\|[0-9][tT][hH]\\)" )
            ((street))
            ((title))))
          ((pp.name matches "[1-9][0-9]+")
            ((street))
            ((title))))))))
```

Note that these only assign values for the feature `token_pos` and do nothing more. You must have a related token to word rule that interprets this feature value and does the required translation. For example the corresponding token to word rule for the above disambiguator is

```

((string-matches name "\\([dD][Rr]\\|[Ss][tT]\\)")
 (if (string-equal (item.feats token "token_pos") "street")
     (if (string-matches name "[dD][rR]")
         (list "drive")
         (list "street"))
     (if (string-matches name "[dD][rR]")
         (list "doctor")
         (list "saint"))))

```

[\[<\]](#)
[\[>\]](#)
[\[<<\]](#)
[\[Up\]](#)
[\[>>\]](#)
[\[Top\]](#)
[\[Contents\]](#)
[\[Index\]](#)
[\[?\]](#)

15.3.2 Building disambiguators

Festival offers some support for building disambiguation trees. The basic method is to find all occurrences of a homographic token in a large text database, label each occurrence into classes, extract appropriate context features for these tokens and finally build an classification tree or decision list based on the extracted features.

The extraction and building of trees is not yet a fully automated process in Festival but the file ``festival/examples/toksearch.scm'` shows some basic Scheme code we use for extracting tokens from very large collections of text.

The function `extract_tokens` does the real work. It reads the given file, token by token into a token stream. Each token is tested against the desired tokens and if there is a match the named features are extracted. The token stream will be extended to provide the necessary context. Note that only some features will make any sense in this situation. There is only a token relation so referring to words, syllables etc. is not productive.

In this example databases are identified by a file that lists all the files in the text databases. Its name is expected to be ``bin/DBNAME.files'` where `DBNAME` is the name of the database. The file should contain a list of filenames in the database e.g for the Gutenberg texts the file ``bin/Gutenberg.files'` contains

```

gutenberg/etext90/bill11.txt
gutenberg/etext90/const11.txt
gutenberg/etext90/getty11.txt
gutenberg/etext90/jfk11.txt
...

```

Extracting the tokens is typically done in two passes. The first pass extracts the context (I've used 5 tokens either side). It extracts the file and position, so the token is identified, and the word in context.

Next those examples should be labelled with a small set of classes which identify the type of the token. For example for a token like "Dr" whether it is a person's title or a street identifier. Note that hand-labelling can be laborious, though it is surprising how few tokens of particular types actually exist in 62 million words.

The next task is to extract the tokens with the features that will best distinguish the particular token. In our "Dr" case this will involve punctuation around the token, capitalisation of surrounding tokens etc. After extracting the distinguishing tokens you must line up the labels with these extracted features. It would be easier to extract both the context and the desired features at the same time but experience shows that in labelling, more appropriate features come to mind that will distinguish classes better and you don't want to have to label twice.

Once a set of features consisting of the label and features is created it is easy to use ``wagon'` to create the corresponding decision tree or decision list. ``wagon'` supports both decision trees and decision lists, it may be worth experimenting to find out which give the best results on some held out test data. It appears that decision trees are typically better, but are often much larger, and the size does not always justify the the sometimes only slightly better results.

16. POS tagging

Part of speech tagging is a fairly well-defined process. Festival includes a part of speech tagger following the HMM-type taggers as found in the Xerox tagger and others (e.g. *DeRose88*). Part of speech tags are assigned, based on the probability distribution of tags given a word, and from ngrams of tags. These models are externally specified and a Viterbi decoder is used to assign part of speech tags at run time.

So far this tagger has only been used for English but there is nothing language specific about it. The module `POS` assigns the tags. It accesses the following variables for parameterization.

`pos_lex_name`

The name of a "lexicon" holding reverse probabilities of words given a tag (indexed by word). If this is unset or has the value `NIL` no part of speech tagging takes place.

`pos_ngram_name`

The name of a loaded ngram model of part of speech tags (loaded by `ngram.load`).

`pos_p_start_tag`

The name of the most likely tag before the start of an utterance. This is typically the tag for sentence final punctuation marks.

`pos_pp_start_tag`

The name of the most likely tag two before the start of an utterance. For English this is typically a simple noun, but for other languages it might be a verb. If the ngram model is bigger than three this tag is effectively repeated for the previous left contexts.

`pos_map`

We have found that it is often better to use a rich tagset for prediction of part of speech tags but that in later use (phrase breaks and dictionary lookup) a much more constrained tagset is better. Thus mapping of the predicted tagset to a different tagset is supported. `pos_map` should be a list of pairs consisting of a list of tags to be mapped and the new tag they are to be mapped to.

Note it is important to have the part of speech tagger match the tags used in later parts of the system, particularly the lexicon. Only two of our lexicons used so far have (mappable) part of speech labels.

An example of the part of speech tagger for English can be found in ``lib/pos.scm'`.

17. Phrase breaks

There are two methods for predicting phrase breaks in Festival, one simple and one sophisticated. These two methods are selected through the parameter `Phrase_Method` and phrasing is achieved by the module `Phrasify`.

The first method is by CART tree. If parameter `Phrase_Method` is `cart_tree`, the CART tree in the variable `phrase_cart_tree` is applied to each word to see if a break should be inserted or not. The tree should predict categories `BB` (for big break), `B` (for break) or `NB` (for no break). A simple example of a tree to predict phrase breaks is given in the file ``lib/phrase.scm'`.

```
(set! simple_phrase_cart_tree
,
((R:Token.parent.punc in ("?" "." ":"))
 ((BB))
 (R:Token.parent.punc in ("'" "\" " "," ";"))
 ((B))
 ((n.name is 0)
 ((BB))
 ((NB))))))
```

The second and more elaborate method of phrase break prediction is used when the parameter `Phrase_Method` is `prob_models`. In this case a probabilistic model using probabilities of a break after a word based on the part of speech of the neighbouring words and the previous word. This is combined with a ngram model of the distribution of breaks and non-breaks using a Viterbi decoder to find the optimal phrasing of the utterance. The results using this technique are good and even show good results on unseen data from other researchers' phrase break tests (see *black97b*). However sometimes it does sound wrong, suggesting there is still further work required.

Parameters for this module are set through the feature list held in the variable `phr_break_params`, and example of which for English is set in `english_phr_break_params` in the file ``lib/phrase.scm'`. The features names and meaning are

```
pos_ngram_name
    The name of a loaded ngram that gives probability distributions of B/NB given previous, current and next part
    of speech.
pos_ngram_filename
    The filename containing pos_ngram_name.
break_ngram_name
    The name of a loaded ngram of B/NB distributions. This is typically a 6 or 7-gram.
break_ngram_filename
    The filename containing break_ngram_name.
gram_scale_s
    A weighting factor for breaks in the break/non-break ngram. Increasing the value insertes more breaks,
    reducing it causes less breaks to be inserted.
phrase_type_tree
    A CART tree that is used to predict type of break given the predict break position. This (rather crude)
    technique is current used to distinguish major and minor breaks.
break_tags
    A list of the break tags (typically (B NB)).
pos_map
    A part of speech map used to map the pos feature of words into a smaller tagset used by the phrase predictor.
```

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

18. Intonation

A number of different intonation modules are available with varying levels of control. In general intonation is generated in two steps.

1. Prediction of accents (and/or end tones) on a per syllable basis.
2. Prediction of F0 target values, this must be done after durations are predicted.

Reflecting this split there are two main intonation modules that call sub-modules depending on the desired intonation methods. The `Intonation` and `Int_Targets` modules are defined in Lisp (``lib/intonation.scm'`) and call sub-modules which are (so far) in C++.

18.1 Default intonation	Effectively none at all.
18.2 Simple intonation	Accents and hats.
18.3 Tree intonation	Accents and Tones, and F0 prediction by LR
18.4 Tilt intonation	Using the Tilt intonation model
18.5 General intonation	A programmable intonation module
18.6 Using ToBI	A ToBI by rule example

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

18.1 Default intonation

This is the simplest form of intonation and offers the modules `Intonation_Default` and `Intonation_Targets_Default`. The first of which actually does nothing at all. `Intonation_Targets_Default` simply creates a target at the start of the utterance, and one at the end. The values of which, by default are 130 Hz and 110 Hz. These values may be set through the parameter `duffint_params` for example the following will general a monotone at 150Hz.

```
(set! duffint_params '((start 150) (end 150)))
(Parameter.set 'Int_Method 'DuffInt)
(Parameter.set 'Int_Target_Method Int_Targets_Default)
```

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

18.2 Simple intonation

This module uses the CART tree in `int_accent_cart_tree` to predict if each syllable is accented or not. A predicted value of `NONE` means no accent is generated by the corresponding `Int_Targets_Simple` function. Any other predicted value will cause a `hat' accent to be put on that syllable.

A default `int_accent_cart_tree` is available in the value `simple_accent_cart_tree` in ``lib/intonation.scm'`. It simply predicts accents on the stressed syllables on content words in poly-syllabic words, and on the only syllable in single syllable content words. Its form is

```
(set! simple_accent_cart_tree
  '
  ((R:SylStructure.parent.gpos is content)
   ((stress is 1)
    ((Accented)
     ((position_type is single)
      ((Accented)
       ((NONE))))
    ((NONE))))
  ((NONE))))
```

The function `Int_Targets_Simple` uses parameters in the a-list in variable `int_simple_params`. There are two interesting parameters `f0_mean` which gives the mean F0 for this speaker (default 110 Hz) and `f0_std` is the standard deviation of F0 for this speaker (default 25 Hz). This second value is used to determine the amount of variation to be put in the generated targets.

For each Phrase in the given utterance an F0 is generated starting at `f0_code+(f0_std*0.6)` and declines `f0_std` Hz over the length of the phrase until the last syllable whose end is set to `f0_code-f0_std`. An imaginary line called `baseline` is drawn from start to the end (minus the final extra fall), For each syllable that is

accented (i.e. has an `IntEvent` related to it) three targets are added. One at the start, one in mid vowel, and one at the end. The start and end are at position `baseline` Hz (as declined for that syllable) and the mid vowel is set to `baseline+f0_std`.

Note this model is not supposed to be complex or comprehensive but it offers a very quick and easy way to generate something other than a fixed line F0. Something similar to this has been for Spanish and Welsh without (too many) people complaining. However it is not designed as a serious intonation module.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

18.3 Tree intonation

This module is more flexible. Two different CART trees can be used to predict `accents' and `endtones'. Although at present this module is used for an implementation of the ToBI intonation labelling system it could be used for many different types of intonation system.

The target module for this method uses a Linear Regression model to predict start mid-vowel and end targets for each syllable using arbitrarily specified features. This follows the work described in *black96*. The LR models are held as described below See section [25.5 Linear regression](#). Three models are used in the variables `f0_lr_start`, `f0_lr_mid` and `f0_lr_end`.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

18.4 Tilt intonation

Tilt description to be inserted.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

18.5 General intonation

As there seems to be a number of intonation theories that predict F0 contours by rule (possibly using trained parameters) this module aids the external specification of such rules for a wide class of intonation theories (through primarily those that might be referred to as the ToBI group). This is designed to be multi-lingual and offer a quick way to port often pre-existing rules into Festival without writing new C++ code.

The accent prediction part uses the same mechanisms as the Simple intonation method described above, a decision tree for accent prediction, thus the tree in the variable `int_accent_cart_tree` is used on each syllable to predict an `IntEvent`.

The target part calls a specified Scheme function which returns a list of target points for a syllable. In this way any arbitrary tests may be done to produce the target points. For example here is a function which returns three target points for each syllable with an `IntEvent` related to it (i.e. accented syllables).

```
(define (targ_func1 utt syl)
  "(targ_func1 UTT STREAMITEM)
Returns a list of targets for the given syllable."
  (let ((start (item.feats syl 'syllable_start))
        (end (item.feats syl 'syllable_end)))
    (if (equal? (item.feats syl "R: Intonation.daughter1.name") "Accented")
        (list
         (list start 110)
         (list (/ (+ start end) 2.0) 140)
         (list end 100))))))
```

This function may be identified as the function to call by the following setup parameters.

```
(Parameter.set 'Int_Method 'General)
(Parameter.set 'Int_Target_Method Int_Targets_General)

(set! int_general_params
  (list
   (list 'targ_func targ_func1)))
```

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

18.6 Using ToBI

An example implementation of a ToBI to F0 target module is included in ``lib/tobi_rules.scm'` based on the rules described in *jilka96*. This uses the general intonation method discussed in the previous section. This is designed to be useful to people who are experimenting with ToBI (*silverman92*), rather than general text to speech.

To use this method you need to load ``lib/tobi_rules.scm'` and call `setup_tobi_f0_method`. The default is in a male's pitch range, i.e. for `voice_rab_diphone`. You can change it for other pitch ranges by changing the following variables.

```
(Parameter.set 'Default_Toplevel 110)
(Parameter.set 'Default_Start_Baseline 87)
(Parameter.set 'Default_End_Baseline 83)
(Parameter.set 'Current_Toplevel (Parameter.get 'Default_Toplevel))
(Parameter.set 'Valley_Dip 75)
```

An example using this from STML is given in ``examples/tobi.stml'`. But it can also be used from Scheme. For example before defining an utterance you should execute the following either from the command line or in some setup file

```
(voice_rab_diphone)
(require 'tobi_rules)
(setup_tobi_f0_method)
```

In order to allow specification of accents, tones, and break levels you must use an utterance type that allows such specification. For example

```
(Utterance
Words
(boy
(saw ((accent H*))
the
(girl ((accent H*))
in the
(park ((accent H*) (tone H-)))
with the
(telescope ((accent H*) (tone H-H%))))))
```

```
(Utterance Words
(The
(boy ((accent L*))
saw
the
(girl ((accent H*) (tone L-)))
with
the
(telescope ((accent H*) (tone H-H%))))))
```

You can display the synthesized form of these utterance in Xwaves. Start an Xwaves and an Xlabeller and call the function `display` on the synthesized utterance.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

19. Duration

A number of different duration prediction modules are available with varying levels of sophistication.

Segmental duration prediction is done by the module `Duration` which calls different actual methods depending on the parameter `Duration_Method`.

All of the following duration methods may be further affected by both a global duration stretch and a per word one.

If the parameter `Duration_Stretch` is set, all absolute durations predicted by any of the duration methods described here are multiplied by the parameter's value. For example

```
(Parameter.set 'Duration_Stretch 1.2)
```

will make everything speak more slowly.

In addition to the global stretch method, if the feature `dur_stretch` on the related `Token` is set it will also be used as a multiplicative factor on the duration produced by the selected method. That is `R:Syllable.parent.parent.R:Token.parent.dur_stretch`. There is a lisp function `duration_find_stretch` which will return the combined global and local duration stretch factor for a given segment item.

Note these global and local methods of affecting the duration produced by models are crude and should be considered hacks. Uniform modification of durations is not what happens in real speech. These parameters are typically used when the underlying duration method is lacking in some way. However these can be useful.

Note it is quite easy to implement new duration methods in Scheme directly.

[19.1 Default durations](#) Fixed length durations

[19.2 Average durations](#)

[19.3 Klatt durations](#) Klatt rules from book.

[19.4 CART durations](#) Tree based durations

[<](#) [>](#) [<<](#) [Up](#) [>>](#) [Top](#) [Contents](#) [Index](#) [?](#)

19.1 Default durations

If parameter `Duration_Method` is set to `Default`, the simplest duration model is used. All segments are 100 milliseconds (this can be modified by `Duration_Stretch`, and/or the localised Token related `dur_stretch` feature).

[<](#) [>](#) [<<](#) [Up](#) [>>](#) [Top](#) [Contents](#) [Index](#) [?](#)

19.2 Average durations

If parameter `Duration_Method` is set to `Averages` then segmental durations are set to their averages. The variable `phoneme_durations` should be an a-list of phones and averages in seconds. The file ``lib/mrpa_durs.scm'` has an example for the `mrpa` phoneset.

If a segment is found that does not appear in the list a default duration of 0.1 seconds is assigned, and a warning message generated.

[<](#) [>](#) [<<](#) [Up](#) [>>](#) [Top](#) [Contents](#) [Index](#) [?](#)

19.3 Klatt durations

If parameter `Duration_Method` is set to `Klatt` the duration rules from the Klatt book (*allen87*, chapter 9). This method requires minimum and inherent durations for each phoneme in the phoneset. This information is held in the variable `duration_klatt_params`. Each member of this list is a three-tuple, of phone name, inherent duration and minimum duration. An example for the `mrpa` phoneset is in ``lib/klatt_durs.scm'`.

[<](#) [>](#) [<<](#) [Up](#) [>>](#) [Top](#) [Contents](#) [Index](#) [?](#)

19.4 CART durations

Two very similar methods of duration prediction by CART tree are supported. The first, used when parameter `Duration_Method` is `Tree` simply predicts durations directly for each segment. The tree is set in the variable `duration_cart_tree`.

The second, which seems to give better results, is used when parameter `Duration_Method` is `Tree_ZScores`. In this second model the tree predicts zscores (number of standard deviations from the mean) rather than duration directly. (This follows *campbell91*, but we don't deal in syllable durations here.) This method requires means and standard deviations for each phone. The variable `duration_cart_tree` should contain the zscore prediction tree and the variable `duration_ph_info` should contain a list of phone, mean duration, and standard deviation for each phone in the phoneset.

An example tree trained from 460 sentences spoken by Gordon is in ``lib/gswdurtreeZ'`. Phone means and standard deviations are in ``lib/gsw_durs.scm'`.

After prediction the segmental duration is calculated by the simple formula

```
duration = mean + (zscore * standard deviation)
```

For some other duration models that affect an inherent duration by some factor this method has been used. If the tree predicts factors rather than zscores and the `duration_ph_info` entries are phone, 0.0, inherent duration. The above formula will generate the desired result. Klatt and Klatt-like rules can be implemented in the this way without adding a new method.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

20. UniSyn synthesizer

Since 1.3 a new general synthesizer module has been included. This designed to replace the older diphone synthesizer described in the next chapter. A redesign was made in order to have a generalized waveform synthesizer, singla processing module that could be used even when the units being concatenated are not diphones. Also at this stage the full diphone (or other) database pre-processing functions were added to the Speech Tool library.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

20.1 UniSyn database format

The Unisyn synthesis modules can use databases in two basic formats, *separate* and *grouped*. Separate is when all files (signal, pitchmark and coefficient files) are accessed individually during synthesis. This is the standard use during database development. Group format is when a database is collected together into a single special file containing all information necessary for waveform synthesis. This format is designed to be used for distribution and general use of the database.

A database should consist of a set of waveforms, (which may be translated into a set of coefficients if the desired the signal processing method requires it), a set of pitchmarks and an index. The pitchmarks are necessary as most of our current signal processing are pitch synchronous.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

20.1.1 Generating pitchmarks

Pitchmarks may be derived from laryngograph files using the our proved program ``pitchmark'` distributed with the speech tools. The actual parameters to this program are still a bit of an art form. The first major issue is which direction the lar files. We have seen both, though it does seem to be CSTR's ones are most often upside down while others (e.g. OGI's) are the right way up. The `-inv` argument to ``pitchmark'` is specifically provided to cater for this. There other issues in getting the pitchmarks aligned. The basic command for generating pitchmarks is

```
pitchmark -inv lar/file001.lar -o pm/file001.pm -otype est \  
-min 0.005 -max 0.012 -fill -def 0.01 -wave_end
```

The ``-min'`, ``-max'` and ``-def'` (fill values for unvoiced regions), may need to be changed depending on the speaker pitch range. The above is suitable for a male speaker. The ``-fill'` option states that unvoiced sections should be filled with equally spaced pitchmarks.

20.1.2 Generating LPC coefficients

LPC coefficients are generated using the `sig2fv` command. Two stages are required, generating the LPC coefficients and generating the residual. The prototypical commands for these are

```
sig2fv wav/file001.wav -o lpc/file001.lpc -otype est -lpc_order 16 \  
-coefs "lpc" -pm pm/file001.pm -preemph 0.95 -factor 3 \  
-window_type hamming  
sigfilter wav/file001.wav -o lpc/file001.res -otype nist \  
-lpcfilter lpc/file001.lpc -inv_filter
```

For some databases you may need to normalize the power. Properly normalizing power is difficult but we provide a simple function which may do the jobs acceptably. You should do this on the waveform before lpc analysis (and ensure you also do the residual extraction on the normalized waveform rather than the original.

```
ch_wave -scaleN 0.5 wav/file001.wav -o file001.Nwav
```

This normalizes the power by maximizing the signal first then multiplying it by the given factor. If the database waveforms are clean (i.e. no clicks) this can give reasonable results.

20.2 Generating a diphone index

The diphone index consists of a short header following by an ascii list of each diphone, the file it comes from followed by its start middle and end times in seconds. For most databases this files needs to be generated by some database specific script.

An example header is

```
EST_File index  
DataType ascii  
NumEntries 2005  
IndexName rab_diphone  
EST_Header_End
```

The most notable part is the number of entries, which you should note can get out of sync with the actual number of entries if you hand edit entries. I.e. if you add an entry and the system still can't find it check that the number of entries is right.

The entries themselves may take on one of two forms, full entries or index entries. Full entries consist of a diphone name, where the phones are separated by "-"; a file name which is used to index into the pitchmark, LPC and waveform file; and the start, middle (change over point between phones) and end of the phone in the file in seconds of the diphone. For example

```
r-uh    edx_1001    0.225    0.261    0.320  
r-e     edx_1002    0.224    0.273    0.326  
r-i     edx_1003    0.240    0.280    0.321  
r-o     edx_1004    0.212    0.253    0.320
```

The second form of entry is an index entry which simply states that reference to that diphone should actually be made to another. For example

aa-11 &aa-1

This states that the diphone aa-11 should actually use the diphone aa-1. Note they are a number of ways to specify alternates for missing diphones and this method is best used for fixing single or small classes of missing or broken diphones. Index entries may appear anywhere in the file but can't be nested.

Some checks are made one reading this index to ensure times etc are reasonable but multiple entries for the same diphone are not, in that case the later one will be selected.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

20.3 Database declaration

There two major types of database *grouped* and *ungrouped*. Grouped databases come as a single file containing the diphone index, coefficients and residuals for the diphones. This is the standard way databases are distributed as voices in Festival. Ungrouped access diphones from individual files and is designed as a method for debugging and testing databases before distribution. Using ungrouped database is slower but allows quicker changes to the index, and associated coefficient files and residuals without rebuilding the group file.

A database is declared to the system through the command `us_diphone_init`. This function takes a parameter list of various features used for setting up a database. The features are

`name`

An atomic name for this database, used in selecting it from the current set of loaded database.

`index_file`

A filename name containing either a diphone index, as described above, or a group file. The feature `grouped` defines the distinction between this being a group of simple index file.

`grouped`

Takes the value "true" or "false". This defined simple index or if the index file is a grouped file.

`coef_dir`

The directory containing the coefficients, (LPC or just pitchmarks in the PSOLA case).

`sig_dir`

The directory containing the signal files (residual for LPC, full waveforms for PSOLA).

`coef_ext`

The extension for coefficient files, typically ".lpc" for LPC file and ".pm" for pitchmark files.

`sig_ext`

The extension for signal files, typically ".res" for LPC residual files and ".wav" for waveform files.

`default_diphone`

The diphone to be used when the requested one doesn't exist. No matter how careful you are you should always include a default diphone for distributed diphone database. Synthesis will throw an error if no diphone is found and there is no default. Although it is usually an error when this is required it's better to fill in something than stop synthesizing. Typical values for this are silence to silence or schwa to schwa.

`alternates_left`

A list of pairs showing the alternate phone names for the left phone in a diphone pair. This list is used to rewrite the diphone name when the directly requested one doesn't exist. This is the recommended method for dealing with systematic holes in a diphone database.

`alternates_right`

A list of pairs showing the alternate phone names for the right phone in a diphone pair. This list is used to rewrite the diphone name when the directly requested one doesn't exist. This is the recommended method for dealing with systematic holes in a diphone database.

An example database definition is


```
(set! rab_diphone_dir "/projects/festival/lib/voices/english/rab_diphone")
(set! rab_lpc_group
  (list
    '(name "rab_lpc_group")
    (list 'index_file
      (path-append rab_diphone_dir "group/rablpc16k.group"))
    '(alternates_left ((i ii) (ll l) (u uu) (i@ ii) (uh @) (a aa)
                      (u@ uu) (w @) (o oo) (e@ ei) (e ei)
                      (r @)))
    '(alternates_right ((i ii) (ll l) (u uu) (i@ ii)
                       (y i) (uh @) (r @) (w @)))
    '(default_diphone @-@@)
    '(grouped "true")))
(us_dipohone_init rab_lpc_group)
```

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

20.4 Making groupfiles

The function `us_make_group_file` will make a group file of the currently selected US diphone database. It loads in all diphones in the database and saves them in the named file. An optional second argument allows specification of how the group file will be saved. These options are as a feature list. There are three possible options

`track_file_format`

The format for the coefficient files. By default this is `est_binary`, currently the only other alternative is `est_ascii`.

`sig_file_format`

The format for the signal parts of the of the database. By default this is `snd` (Sun's Audio format). This was chosen as it has the smallest header and supports various sample formats. Any format supported by the Edinburgh Speech Tools is allowed.

`sig_sample_format`

The format for the samples in the signal files. By default this is `mulaw`. This is suitable when the signal files are LPC residuals. LPC residuals have a much smaller dynamic range than plain PCM files. Because `mulaw` representation is half the size (8 bits) of standard PCM files (16bits) this significantly reduces the size of the group file while only marginally altering the quality of synthesis (and from experiments the effect is not perceptible). However when saving group files where the signals are not LPC residuals (e.g. in PSOLA) using this default `mulaw` is not recommended and `short` should probably be used.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

20.5 UniSyn module selection

In a voice selection a UniSyn database may be selected as follows

```
(set! UniSyn_module_hooks (list rab_diphone_const_clusters ))
(set! us_abs_offset 0.0)
(set! window_factor 1.0)
(set! us_rel_offset 0.0)
(set! us_gain 0.9)

(Parameter.set 'Synth_Method 'UniSyn)
(Parameter.set 'us_sigpr 'lpc)
(us_db_select rab_db_name)
```

The `UniSyn_module_hooks` are run before synthesis, see the next selection about diphone name selection. At present only `lpc` is supported by the UniSyn module, though potentially there may be others.

An optional implementation of TD-PSOLA *moulines90* has been written but fear of legal problems unfortunately prevents it being in the public distribution, but this policy should not be taken as acknowledging or not acknowledging any alleged patent violation.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

20.6 Diphone selection

Diphone names are constructed for each phone-phone pair in the Segment relation in an utterance. If a segment has the feature in forming a diphone name UniSyn first checks for the feature `us_diphone_left` (or `us_diphone_right` for the right hand part of the diphone) then if that doesn't exist the feature `us_diphone` then if that doesn't exist the feature name. Thus it is possible to specify diphone names which are not simply the concatenation of two segment names.

This feature is used to specify consonant cluster diphone names for our English voices. The hook `UniSyn_module_hooks` is run before selection and we specify a function to add `us_diphone_*` features as appropriate. See the function `rab_diphone_fix_phone_name` in ``lib/voices/english/rab_diphone/festvox/rab_diphone.scm'` for an example.

Once the diphone name is created it is used to select the diphone from the database. If it is not found the name is converted using the list of `alternates_left` and `alternates_right` as specified in the database declaration. If that doesn't specify a diphone in the database. The `default_diphone` is selected, and a warning is printed. If no default diphone is specified or the default diphone doesn't exist in the database an error is thrown.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

21. Diphone synthesizer

NOTE: use of this diphone synthesis is deprecated and it will probably be removed from future versions, all of its functionality has been replaced by the UniSyn synthesizer. It is not compiled by default, if required add `ALSO_INCLUDE += diphone` to your ``festival/config/config'` file.

A basic diphone synthesizer offers a method for making speech from segments, durations and intonation targets. This module was mostly written by Alistair Conkie but the base diphone format is compatible with previous CSTR diphone synthesizers.

The synthesizer offers residual excited LPC based synthesis (*hunt89*) and PSOLA (TM) (*moulines90*) (PSOLA is not available for distribution).

21.1 Diphone database format	Format of basic dbs
21.2 LPC databases	Building and using LPC files.
21.3 Group files	Efficient binary formats
21.4 Diphone Init	Loading diphone databases
21.5 Access strategies	Various access methods
21.6 Diphone selection	Mapping phones to special diphone names

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

21.1 Diphone database format

A diphone database consists of a *dictionary file*, a set of *waveform files*, and a set of *pitch mark files*. These files are the same format as the previous CSTR (Osprey) synthesizer.

The dictionary file consist of one entry per line. Each entry consists of five fields: a diphone name of the form *P1-P2*, a filename (without extension), a floating point start position in the file in milliseconds, a mid position in milliseconds (change in phone), and an end position in milliseconds. Lines starting with a semi-colon and blank lines are ignored. The list may be in any order.

For example a partial list of phones may look like.

ch-l	r021	412.035	463.009	518.23
jh-l	d747	305.841	382.301	446.018
h-l	d748	356.814	403.54	437.522
#-@	d404	233.628	297.345	331.327
@-#	d001	836.814	938.761	1002.48

Waveform files may be in any form, as long as every file is the same type, headered or unheadered as long as the format is supported the speech tools wave reading functions. These may be standard linear PCM waveform files in the case of PSOLA or LPC coefficients and residual when using the residual LPC synthesizer. [21.2 LPC databases](#)

Pitch mark files consist a simple list of positions in milliseconds (plus places after the point) in order, one per line of each pitch mark in the file. For high quality diphone synthesis these should be derived from laryngograph data. During unvoiced sections pitch marks should be artificially created at reasonable intervals (e.g. 10 ms). In the current format there is no way to determine the "real" pitch marks from the "unvoiced" pitch marks.

It is normal to hold a diphone database in a directory with a number of sub-directories namely ``dic/`` contain the dictionary file, ``wave/`` for the waveform files, typically of whole nonsense words (sometimes this directory is called ``vox/`` for historical reasons) and ``pm/`` for the pitch mark files. The filename in the dictionary entry should be the same for waveform file and the pitch mark file (with different extensions).

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

21.2 LPC databases

The standard method for diphone resynthesis in the released system is residual excited LPC (*hunt89*). The actual method of resynthesis isn't important to the database format, but if residual LPC synthesis is to be used then it is necessary to make the LPC coefficient files and their corresponding residuals.

Previous versions of the system used a "host of hacky little scripts" to this but now that the Edinburgh Speech Tools supports LPC analysis we can provide a walk through for generating these.

We assume that the waveform file of nonsense words are in a directory called ``wave/``. The LPC coefficients and residuals will be, in this example, stored in ``lpc16k/`` with extensions ``.lpc`` and ``.res`` respectively.

Before starting it is worth considering power normalization. We have found this important on all of the databases we have collected so far. The `ch_wave` program, part of the speech tools, with the optional `-scaleN 0.4` may be used if a more complex method is not available.

The following shell command generates the files

```

for i in wave/*.wav
do
    fname=`basename $i .wav`
    echo $i
    lpc_analysis -reflection -shift 0.01 -order 18 -o lpc16k/$fname.lpc \
        -r lpc16k/$fname.res -otype htk -rtype nist $i
done

```

It is said that the LPC order should be sample rate divided by one thousand plus 2. This may or may not be appropriate and if you are particularly worried about the database size it is worth experimenting.

The program ``lpc_analysis``, found in ``speech_tools/bin``, can be used to generate the lpc coefficients and residual. Note these should be reflection coefficients so they may be quantised (as they are in group files).

The coefficients and residual files produced by different LPC analysis programs may start at different offsets. For example the Entropic's ESPS functions generate LPC coefficients that are offset by one frame shift (e.g. 0.01 seconds). Our own ``lpc_analysis`` routine has no offset. The `Diphone_Init` parameter list allows these offsets to be specified. Using the above function to generate the LPC files the description parameters should include

```

(lpc_frame_offset 0)
(lpc_res_offset 0.0)

```

While when generating using ESPS routines the description should be

```

(lpc_frame_offset 1)
(lpc_res_offset 0.01)

```

The defaults actually follow the ESPS form, that is `lpc_frame_offset` is 1 and `lpc_res_offset` is equal to the frame shift, if they are not explicitly mentioned.

Note the biggest problem we have in implementing the residual excited LPC resynthesizer was getting the right part of the residual to line up with the right LPC coefficients describing the pitch mark. Making errors in this degrades the synthesized waveform notably, but not seriously, making it difficult to determine if it is an offset problem or some other bug.

Although we have started investigating if extracting pitch synchronous LPC parameters rather than fixed shift parameters gives better performance, we haven't finished this work. ``lpc_analysis`` supports pitch synchronous analysis but the raw "ungrouped" access method does not yet. At present the LPC parameters are extracted at a particular pitch mark by interpolating over the closest LPC parameters. The "group" files hold these interpolated parameters pitch synchronously.

The American English voice ``kd`` was created using the speech tools ``lpc_analysis`` program and its set up should be looked at if you are going to copy it. The British English voice ``rb`` was constructed using ESPS routines.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

21.3 Group files

Databases may be accessed directly but this is usually too inefficient for any purpose except debugging. It is expected that *group files* will be built which contain a binary representation of the database. A group file is a compact efficient representation of the diphone database. Group files are byte order independent, so may be shared between machines of different byte orders and word sizes. Certain information in a group file may be changed at load time so a database name, access strategy etc. may be changed from what was set originally in the group file.

A group file contains the basic parameters, the diphone index, the signal (original waveform or LPC residual), LPC coefficients, and the pitch marks. It is all you need for a run-time synthesizer. Various compression mechanisms are

supported to allow smaller databases if desired. A full English LPC plus residual database at 8k ulaw is about 3 megabytes, while a full 16 bit version at 16k is about 8 megabytes.

Group files are created with the `Diphone.group` command which takes a database name and an output filename as an argument. Making group files can take some time especially if they are large. The `group_type` parameter specifies `raw` or `ulaw` for encoding signal files. This can significantly reduce the size of databases.

Group files may be partially loaded (see access strategies) at run time for quicker start up and to minimise run-time memory requirements.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

21.4 Diphone_Init

The basic method for describing a database is through the `Diphone_Init` command. This function takes a single argument, a list of pairs of parameter name and value. The parameters are

`name`
An atomic name for this database.

`group_file`
The filename of a group file, which may itself contain parameters describing itself

`type`
The default value is `pcm`, but for distributed voices this is always `lpc`.

`index_file`
A filename containing the diphone dictionary.

`signal_dir`
A directory (slash terminated) containing the pcm waveform files.

`signal_ext`
A dot prefixed extension for the pcm waveform files.

`pitch_dir`
A directory (slash terminated) containing the pitch mark files.

`pitch_ext`
A dot prefixed extension for the pitch files

`lpc_dir`
A directory (slash terminated) containing the LPC coefficient files and residual files.

`lpc_ext`
A dot prefixed extension for the LPC coefficient files

`lpc_type`
The type of LPC file (as supported by the speech tools)

`lpc_frame_offset`
The number of frames "missing" from the beginning of the file. Often LPC parameters are offset by one frame.

`lpc_res_ext`
A dot prefixed extension for the residual files

`lpc_res_type`
The type of the residual files, this is a standard waveform type as supported by the speech tools.

`lpc_res_offset`
Number of seconds "missing" from the beginning of the residual file. Some LPC analysis technique do not generate a residual until after one frame.

`samp_freq`
Sample frequency of signal files

`phoneset`
Phoneset used, must already be declared.

`num_diphones`
Total number of diphones in database. If specified this must be equal or bigger than the number of entries in the index file. If it is not specified the square of the number of phones in the phoneset is used.

`sig_band`
number of sample points around actual diphone to take from file. This should be larger than any windowing

used on the signal, and/or up to the pitch marks outside the diphone signal.

`alternates_after`

List of pairs of phones stating replacements for the second part of diphone when the basic diphone is not found in the diphone database.

`alternates_before`

List of pairs of phones stating replacements for the first part of diphone when the basic diphone is not found in the diphone database.

`default_diphone`

When unexpected combinations occur and no appropriate diphone can be found this diphone should be used. This should be specified for all diphone databases that are to be robust. We usually use the silence to silence diphone. No matter how carefully you designed your diphone set, conditions when an unknown diphone occur seem to *always* happen. If this is not set and a diphone is requested that is not in the database an error occurs and synthesis will stop.

Examples of both general set up, making group files and general use are in

```
`lib/voices/english/rab_diphone/festvox/rab_diphone.scm'
```

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

21.5 Access strategies

Three basic accessing strategies are available when using diphone databases. They are designed to optimise access time, start up time and space requirements.

`direct`

Load all signals at database init time. This is the slowest startup but the fastest to access. This is ideal for servers. It is also useful for small databases that can be loaded quickly. It is reasonable for many group files.

`dynamic`

Load signals as they are required. This has much faster start up and will only gradually use up memory as the diphones are actually used. Useful for larger databases, and for non-group file access.

`ondemand`

Load the signals as they are requested but free them if they are not required again immediately. This is slower access but requires low memory usage. In group files the re-reads are quite cheap as the database is well cached and a file description is already open for the file.

Note that in group files pitch marks (and LPC coefficients) are always fully loaded (cf. `direct`), as they are typically smaller. Only signals (waveform files or residuals) are potentially dynamically loaded.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

21.6 Diphone selection

The appropriate diphone is selected based on the name of the phone identified in the segment stream. However for better diphone synthesis it is useful to augment the diphone database with other diphones in addition to the ones directly from the phoneme set. For example dark and light l's, distinguishing consonants from their consonant cluster form and their isolated form. There are however two methods to identify this modification from the basic name.

When the diphone module is called the hook `diphone_module_hooks` is applied. That is a function of list of functions which will be applied to the utterance. Its main purpose is to allow the conversion of the basic name into an augmented one. For example converting a basic l into a dark l, denoted by l1. The functions given in `diphone_module_hooks` may set the feature `diphone_phone_name` which if set will be used rather than the name of the segment.

For example suppose we wish to use a dark l (l1) rather than a normal l for all l's that appear in the coda of a syllable. First we would define a function to which identifies this condition and adds the addition feature `diphone_phone_name` identify the name change. The following function would achieve this

```
(define (fix_dark_ls utt)
  "(fix_dark_ls UTT)
  Identify ls in coda position and relabel them as l1."
  (mapcar
   (lambda (seg)
     (if (and (string-equal "l" (item.name seg))
              (string-equal "+" (item.feats seg "p.ph_vc"))
              (item.relation.prev seg "SylStructure"))
         (item.set_feats seg "diphone_phone_name" "l1")))
     (utt.relation.items utt 'Segment))
   utt)
```

Then when we wish to use this for a particular voice we need to add

```
(set! diphone_module_hooks (list fix_dark_ls))
```

in the voice selection function.

For a more complex example including consonant cluster identification see the American English voice `ked' in ``festival/lib/voices/english/ked/festvox/kd_diphone.scm'`. The function `ked_diphone_fix_phone_name` carries out a number of mappings.

The second method for changing a name is during actual look up of a diphone in the database. The list of alternates is given by the `Diphone_Init` function. These are used when the specified diphone can't be found. For example we often allow mappings of dark l, l1 to l as sometimes the dark l diphone doesn't actually exist in the database.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

22. Other synthesis methods

Festival supports a number of other synthesis systems

- [22.1 LPC diphone synthesizer](#) A small LPC synthesizer (Donovan diphones)
- [22.2 MBROLA](#) Interface to MBROLA
- [22.3 Synthesizers in development](#)

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

22.1 LPC diphone synthesizer

A very simple, and very efficient LPC diphone synthesizer using the "donovan" diphones is also supported. This synthesis method is primarily the work of Steve Isard and later Alistair Conkie. The synthesis quality is not as good as the residual excited LPC diphone synthesizer but has the advantage of being much smaller. The donovan diphone database is under 800k.

The diphones are loaded through the `Donovan_Init` function which takes the name of the dictionary file and the diphone file as arguments, see the following for details

22.2 MBROLA

As an example of how Festival may use a completely external synthesis method we support the free system MBROLA. MBROLA is both a diphone synthesis technique and an actual system that constructs waveforms from segment, duration and F0 target information. For details see the MBROLA home page at <http://tcts.fpms.ac.be/synthesis/mbrola.html>. MBROLA already supports a number of diphone sets including French, Spanish, German and Romanian.

Festival support for MBROLA is in the file ``lib/mbrola.scm'`. It is all in Scheme. The function `MBROLA_Synth` is called when parameter `Synth_Method` is MBROLA. The function simply saves the segment, duration and target information from the utterance, calls the external ``mbrola'` program with the selected diphone database, and reloads the generated waveform back into the utterance.

An MBROLA-ized version of the Roger diphoneset is available from the MBROLA site. The simple Festival end is distributed as part of the system in ``festvox_en1.tar.gz'`. The following variables are used by the process

`mbrola_progname`
the pathname of the mbrola executable.
`mbrola_database`
the name of the database to use. This variable is switched between different speakers.

22.3 Synthesizers in development

In addition to the above synthesizers Festival also supports CSTR's older PSOLA synthesizer written by Paul Taylor. But as the newer diphone synthesizer produces similar quality output and is a newer (and hence a cleaner) implementation further development of the older module is unlikely.

An experimental unit selection synthesis module is included in ``modules/clunits/'` it is an implementation of *black97c*. It is included for people wishing to continue research in the area rather than as a fully usable waveform synthesis engine. Although it sometimes gives excellent results it also sometimes gives amazingly bad ones too. We included this as an example of one possible framework for selection-based synthesis.

As one of our funded projects is to specifically develop new selection based synthesis algorithms we expect to include more models within later versions of the system.

Also, now that Festival has been released other groups are working on new synthesis techniques in the system. Many of these will become available and where possible we will give pointers from the Festival home page to them. Particularly there is an alternative residual excited LPC module implemented at the Center for Spoken Language Understanding (CSLU) at the Oregon Graduate Institute (OGI).

23. Audio output

If you have never heard any audio ever on your machine then you must first work out if you have the appropriate hardware. If you do, you also need the appropriate software to drive it. Festival can directly interface with a number of audio systems or use external methods for playing audio.

The currently supported audio methods are

``NAS'`

NCD's NAS, is a network transparent audio system (formerly called netaudio). If you already run servers on your machines you simply need to ensure your AUDIOSERVER environment variable is set (or your DISPLAY variable if your audio output device is the same as your X Windows display). You may set NAS as your audio output method by the command

```
(Parameter.set 'Audio_Method 'netaudio)
```

``/dev/audio'`

On many systems ``/dev/audio'` offers a simple low level method for audio output. It is limited to mu-law encoding at 8KHz. Some implementations of ``/dev/audio'` allow other sample rates and sample types but as that is non-standard this method only uses the common format. Typical systems that offer these are Suns, Linux and FreeBSD machines. You may set direct ``/dev/audio'` access as your audio method by the command

```
(Parameter.set 'Audio_Method 'sunaudio)
```

``/dev/audio (16bit)'`

Later Sun Microsystems workstations support 16 bit linear audio at various sample rates. Support for this form of audio output is supported. It is a compile time option (as it requires include files that only exist on Sun machines. If your installation supports it (check the members of the list `*modules*`) you can select 16 bit audio output on Suns by the command

```
(Parameter.set 'Audio_Method 'sun16audio)
```

Note this will send it to the local machine where the festival binary is running, this might not be the one you are sitting next to--that's why we recommend netaudio. A hacky solution to playing audio on a local machine from a remote machine without using netaudio is described in [6. Installation](#)

``/dev/dsp (voxware)'`

Both FreeBSD and Linux have a very similar audio interface through ``/dev/dsp'`. There is compile time support for these in the speech tools and when compiled with that option Festival may utilise it. Check the value of the variable `*modules*` to see which audio devices are directly supported. On FreeBSD, if supported, you may select local 16 bit linear audio by the command

```
(Parameter.set 'Audio_Method 'freebsd16audio)
```

While under Linux, if supported, you may use the command

```
(Parameter.set 'Audio_Method 'linux16audio)
```

Some earlier (and smaller machines) only have 8bit audio even though they include a ``/dev/dsp'` (Soundblaster PRO for example). This was not dealt with properly in earlier versions of the system but now the support automatically checks to see the sample width supported and uses it accordingly. 8 bit at higher frequencies that 8K sounds better than straight 8k ulaw so this feature is useful.

``mplayer'`

Under Windows NT or 95 you can use the ``mplayer'` command which we have found requires special treatment to get its parameters right. Rather than using `Audio_Command` you can select this on Windows machine with the following command

```
(Parameter.set 'Audio_Method 'mplayeraudio)
```

Alternatively built-in audio output is available with

```
(Parameter.set 'Audio_Method 'win32audio)
```

``SGI IRIX'`

Builtin audio output is now available for SGI's IRIX 6.2 using the command

```
(Parameter.set 'Audio_Method 'irixaudio)
```

`Audio Command'

Alternatively the user can provide a command that can play an audio file. Festival will execute that command in an environment where the shell variables `SR` is set to the sample rate (in Hz) and `FILE` which, by default, is the name of an unheadered raw, 16bit file containing the synthesized waveform in the byte order of the machine Festival is running on. You can specify your audio play command and that you wish Festival to execute that command through the following command

```
(Parameter.set 'Audio_Command "sun16play -f $SR $FILE")  
(Parameter.set 'Audio_Method 'Audio_Command)
```

On SGI machines under IRIX the equivalent would be

```
(Parameter.set 'Audio_Command  
    "sfplay -i integer 16 2scomp rate $SR end $FILE")  
(Parameter.set 'Audio_Method 'Audio_Command)
```

The `Audio_Command` method of playing waveforms Festival supports two additional audio parameters. `Audio_Required_Rate` allows you to use Festival's internal sample rate conversion function to any desired rate. Note this may not be as good as playing the waveform at the sample rate it is originally created in, but as some hardware devices are restrictive in what sample rates they support, or have naive resample functions this could be optimal. The second additional audio parameter is `Audio_Required_Format` which can be used to specify the desired output forms of the file. The default is unheadered raw, but this may be any of the values supported by the speech tools (including `nist`, `esps`, `snd`, `riff`, `aiff`, `audlab`, `raw` and, if you really want it, `ascii`). For example suppose you have a program that only plays sun headered files at 16000 KHz you can set up audio output as

```
(Parameter.set 'Audio_Method 'Audio_Command)  
(Parameter.set 'Audio_Required_Rate 16000)  
(Parameter.set 'Audio_Required_Format 'snd)  
(Parameter.set 'Audio_Command "sunplay $FILE")
```

Where the audio method supports it, you can specify alternative audio device for machine that have more than one audio device.

```
(Parameter.set 'Audio_Device "/dev/dsp2")
```

If Netaudio is not available and you need to play audio on a machine different from the one Festival is running on we have had reports that ``snack'` (<http://www.speech.kth.se/snack/>) is a possible solution. It allows remote play but importantly also supports Windows 95/NT based clients.

Because you do not want to wait for a whole file to be synthesized before you can play it, Festival also offers an *audio spooler* that allows the playing of audio files while continuing to synthesize the following utterances. On reasonable workstations this allows the breaks between utterances to be as short as your hardware allows them to be.

The audio spooler may be started by selecting asynchronous mode

```
(audio_mode 'async)
```

This is switched on by default by the function `tts`. You may put Festival back into synchronous mode (i.e. the `utt.play` command will wait until the audio has finished playing before returning). by the command

```
(audio_mode 'sync)
```

Additional related commands are

```
(audio_mode 'close)
```

Close the audio server down but wait until it is cleared. This is useful in scripts etc. when you wish to only exit when all audio is complete.

(audio_mode 'shutup)

Close the audio down now, stopping the current file being played and any in the queue. Note that this may take some time to take effect depending on which audio method you use. Sometimes there can be 100s of milliseconds of audio in the device itself which cannot be stopped.

(audio_mode 'query)

Lists the size of each waveform currently in the queue.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

24. Voices

This chapter gives some general suggestions about adding new voices to Festival. Festival attempts to offer an environment where new voices and languages can easily be slotted in to the system.

[24.1 Current voices](#) Currently available voices

[24.2 Building a new voice](#)

[24.3 Defining a new voice](#)

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

24.1 Current voices

Currently there are a number of voices available in Festival and we expect that number to increase. Each is elected via a function of the name ``voice_*` which sets up the waveform synthesizer, phone set, lexicon, duration and intonation models (and anything else necessary) for that speaker. These voice setup functions are defined in ``lib/voices.scm'`.

The current voice functions are

`voice_rab_diphone`

A British English male RP speaker, Roger. This uses the UniSyn residual excited LPC diphone synthesizer. The lexicon is the computer users version of Oxford Advanced Learners' Dictionary, with letter to sound rules trained from that lexicon. Intonation is provided by a ToBI-like system using a decision tree to predict accent and end tone position. The F0 itself is predicted as three points on each syllable, using linear regression trained from the Boston University FM database (f2b) and mapped to Roger's pitch range. Duration is predicted by decision tree, predicting zscore durations for segments trained from the 460 Timit sentence spoken by another British male speaker.

`voice_ked_diphone`

An American English male speaker, Kurt. Again this uses the UniSyn residual excited LPC diphone synthesizer. This uses the CMU lexicon, and letter to sound rules trained from it. Intonation as with Roger is trained from the Boston University FM Radio corpus. Duration for this voice also comes from that database.

`voice_kal_diphone`

An American English male speaker. Again this uses the UniSyn residual excited LPC diphone synthesizer. And like ked, uses the CMU lexicon, and letter to sound rules trained from it. Intonation as with Roger is trained from the Boston University FM Radio corpus. Duration for this voice also comes from that database. This voice was built in two days work and is at least as good as ked due to us understanding the process better. The diphone labels were autoaligned with hand correction.

`voice_don_diphone`

Steve Isard's LPC based diphone synthesizer, Donovan diphones. The other parts of this voice, lexicon, intonation, and duration are the same as `voice_rab_diphone` described above. The quality of the diphones is not as good as the other voices because it uses spike excited LPC. Although the quality is not as good it is much faster and the database is much smaller than the others.

voice_el_diphone

A male Castilian Spanish speaker, using the Eduardo Lopez diphones. Alistair Conkie and Borja Etxebarria did much to make this. It has improved recently but is not as comprehensive as our English voices.

voice_gsw_diphone

This offers a male RP speaker, Gordon, famed for many previous CSTR synthesizers, using the standard diphone module. Its higher levels are very similar to the Roger voice above. This voice is not in the standard distribution, and is unlikely to be added for commercial reasons, even though it sounds better than Roger.

voice_en1_mbrola

The Roger diphone set using the same front end as `voice_rab_diphone` but uses the MBROLA diphone synthesizer for waveform synthesis. The MBROLA synthesizer and Roger diphone database (called `en1`) is not distributed by CSTR but is available for non-commercial use for free from <http://tcts.fpms.ac.be/synthesis/mbrola.html>. We do however provide the Festival part of the voice in ``festvox_en1.tar.gz'`.

voice_us1_mbrola

A female American English voice using our standard US English front end and the `us1` database for the MBROLA diphone synthesizer for waveform synthesis. The MBROLA synthesizer and the `us1` diphone database is not distributed by CSTR but is available for non-commercial use for free from <http://tcts.fpms.ac.be/synthesis/mbrola.html>. We provide the Festival part of the voice in ``festvox_us1.tar.gz'`.

voice_us2_mbrola

A male American English voice using our standard US English front end and the `us2` database for the MBROLA diphone synthesizer for waveform synthesis. The MBROLA synthesizer and the `us2` diphone database is not distributed by CSTR but is available for non-commercial use for free from <http://tcts.fpms.ac.be/synthesis/mbrola.html>. We provide the Festival part of the voice in ``festvox_us2.tar.gz'`.

voice_us3_mbrola

Another male American English voice using our standard US English front end and the `us2` database for the MBROLA diphone synthesizer for waveform synthesis. The MBROLA synthesizer and the `us2` diphone database is not distributed by CSTR but is available for non-commercial use for free from <http://tcts.fpms.ac.be/synthesis/mbrola.html>. We provide the Festival part of the voice in ``festvox_us1.tar.gz'`.

Other voices will become available through time. Groups other than CSTR are working on new voices. Particularly OGI's CSLU have release a number of American English voices, two Mexican Spanish voices and two German voices. All use OGI's their own residual excited LPC synthesizer which is distributed as a plug-in for Festival. (see <http://www.cse.ogi.edu/CSLU/research/TTS> for details).

Other languages are being worked on including German, Basque, Welsh, Greek and Polish already have been developed and could be release soon. CSTR has a set of Klingon diphones though the text analysis for Klingon still requires some work (If anyone has access to a good Klingon continuous speech corpora please let us know.)

Pointers and examples of voices developed at CSTR and elsewhere will be posted on the Festival home page.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

24.2 Building a new voice

This section runs through the definition of a new voice in Festival. Although this voice is simple (it is a simplified version of the distributed spanish voice) it shows all the major parts that must be defined to get Festival to speak in a new voice. Thanks go to Alistair Conkie for helping me define this but as I don't speak Spanish there are probably many mistakes. Hopefully its pedagogical use is better than its ability to be understood in Castille.

A much more detailed document on building voices in Festival has been written and is recommend reading for any one attempting to add a new voice to Festival *black99*. The information here is a little sparse though gives the basic requirements.

The general method for defining a new voice is to define the parameters for all the various sub-parts e.g. phoneset,

duration parameter intonation parameters etc., then defined a function of the form `voice_NAME` which when called will actually select the voice.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

24.2.1 Phoneset

For most new languages and often for new dialects, a new phoneset is required. It is really the basic building block of a voice and most other parts are defined in terms of this set, so defining it first is a good start.

```
(defPhoneSet
  spanish
  ;; Phone Features
  ;; vowel or consonant
  (vc + -)
  ;; vowel length: short long diphthong schwa
  (vlngr s l d a 0)
  ;; vowel height: high mid low
  (vheight 1 2 3 -)
  ;; vowel frontness: front mid back
  (vfront 1 2 3 -)
  ;; lip rounding
  (vrnd + -)
  ;; consonant type: stop fricative affricative nasal liquid
  (ctype s f a n l 0)
  ;; place of articulation: labial alveolar palatal labio-dental
  ;;                               dental velar
  (cplace l a p b d v 0)
  ;; consonant voicing
  (cvox + -)
  )
;; Phone set members (features are not! set properly)
(
  (# - 0 - - - 0 0 -)
  (a + 1 3 1 - 0 0 -)
  (e + 1 2 1 - 0 0 -)
  (i + 1 1 1 - 0 0 -)
  (o + 1 3 3 - 0 0 -)
  (u + 1 1 3 + 0 0 -)
  (b - 0 - - + s l +)
  (ch - 0 - - + a a -)
  (d - 0 - - + s a +)
  (f - 0 - - + f b -)
  (g - 0 - - + s p +)
  (j - 0 - - + l a +)
  (k - 0 - - + s p -)
  (l - 0 - - + l d +)
  (ll - 0 - - + l d +)
  (m - 0 - - + n l +)
  (n - 0 - - + n d +)
  (ny - 0 - - + n v +)
  (p - 0 - - + s l -)
  (r - 0 - - + l p +)
  (rr - 0 - - + l p +)
  (s - 0 - - + f a +)
  (t - 0 - - + s t +)
  (th - 0 - - + f d +)
  (x - 0 - - + a a -)
  )
)
```

```
)  
(PhoneSet.silences '#')
```

Note some phonetic features may be wrong.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

24.2.2 Lexicon and LTS

Spanish is a language whose pronunciation can almost completely be predicted from its orthography so in this case we do not need a list of words and their pronunciations and can do most of the work with letter to sound rules.

Let us first make a lexicon structure as follows

```
(lex.create "spanish")  
(lex.set.phoneset "spanish")
```

However if we did just want a few entries to test our system without building any letter to sound rules we could add entries directly to the addenda. For example

```
(lex.add.entry  
  '("amigos" nil ((a) 0) ((m i) 1) (g o s))))
```

A letter to sound rule system for Spanish is quite simple in the format supported by Festival. The following is a good start to a full set.

```
(lts.ruleset  
; Name of rule set  
spanish  
; Sets used in the rules  
(  
  (LNS l n s )  
  (AEOU a e o u )  
  (AEO a e o )  
  (EI e i )  
  (BDGLMN b d g l m n )  
)  
; Rules  
(  
  ( [ a ] = a )  
  ( [ e ] = e )  
  ( [ i ] = i )  
  ( [ o ] = o )  
  ( [ u ] = u )  
  ( [ "" a ] = a1 )    ;; stressed vowels  
  ( [ "" e ] = e1 )  
  ( [ "" i ] = i1 )  
  ( [ "" o ] = o1 )  
  ( [ "" u ] = u1 )  
  ( [ b ] = b )  
  ( [ v ] = b )  
  ( [ c ] "" EI = th )  
  ( [ c ] EI = th )  
  ( [ c h ] = ch )  
  ( [ c ] = k )  
  ( [ d ] = d )  
  ( [ f ] = f )  
  ( [ g ] "" EI = x )  
  ( [ g ] EI = x )
```

```

( [ g u ] "" EI = g )
( [ g u ] EI = g )
( [ g ] = g )
( [ h u e ] = u e )
( [ h i e ] = i e )
( [ h ] = )
( [ j ] = x )
( [ k ] = k )
( [ l l ] # = l )
( [ l l ] = ll )
( [ l ] = l )
( [ m ] = m )
( [ ~ n ] = ny )
( [ n ] = n )
( [ p ] = p )
( [ q u ] = k )
( [ r r ] = rr )
( # [ r ] = rr )
( LNS [ r ] = rr )
( [ r ] = r )
( [ s ] BDGLMN = th )
( [ s ] = s )
( # [ s ] C = e s )
( [ t ] = t )
( [ w ] = u )
( [ x ] = k s )
( AEO [ y ] = i )
( # [ y ] # = i )
( [ y ] = ll )
( [ z ] = th )
))

```

We could simply set our lexicon to use the above letter to sound system with the following command

```
(lex.set.lts.ruleset 'spanish)
```

But this would not deal with upper case letters. Instead of writing new rules for upper case letters we can define that a Lisp function be called when looking up a word and intercept the lookup with our own function. First we state that unknown words should call a function, and then define the function we wish called. The actual link to ensure our function will be called is done below at lexicon selection time

```

(define (spanish_lts word features)
  "(spanish_lts WORD FEATURES)
  Using letter to sound rules build a spanish pronunciation of WORD."
  (list word
        nil
        (lex.syllabify.phstress (lts.apply (downcase word) 'spanish))))
(lex.set.lts.method spanish_lts)

```

In the function we downcase the word and apply the LTS rule to it. Next we syllabify it and return the created lexical entry.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

24.2.3 Phrasing

Without detailed labelled databases we cannot build statistical models of phrase breaks, but we can simply build a phrase break model based on punctuation. The following is a CART tree to predict simple breaks, from punctuation.

```
(set! spanish_phrase_cart_tree
,
((lisp_token_end_punc in ("?" "." ":"))
 ((BB))
 (lisp_token_end_punc in ("'" "\" " ", " ";"))
 ((B))
 ((n.name is 0) ;; end of utterance
 ((BB))
 ((NB))))))
```

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

24.2.4 Intonation

For intonation there are number of simple options without requiring training data. For this example we will simply use a hat pattern on all stressed syllables in content words and on single syllable content words. (i.e. Simple) Thus we need an accent prediction CART tree.

```
(set! spanish_accent_cart_tree
,
((R:SylStructure.parent.gpos is content)
 ((stress is 1)
 ((Accented)
 ((position_type is single)
 ((Accented)
 ((NONE))))))
 ((NONE))))
```

We also need to specify the pitch range of our speaker. We will be using a male Spanish diphone database of the follow range

```
(set! spanish_el_int_simple_params
'((f0_mean 120) (f0_std 30)))
```

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

24.2.5 Duration

We will use the trick mentioned above for duration prediction. Using the zscore CART tree method, we will actually use it to predict factors rather than zscores.

The tree predicts longer durations in stressed syllables and in clause initial and clause final syllables.


```
(set! spanish_dur_tree
'
  ((R:SylStructure.parent.R:Syllable.p.syl_break > 1 ) ;; clause initial
  ((R:SylStructure.parent.stress is 1)
  ((1.5))
  ((1.2)))
  ((R:SylStructure.parent.syl_break > 1) ;; clause final
  ((R:SylStructure.parent.stress is 1)
  ((2.0))
  ((1.5)))
  ((R:SylStructure.parent.stress is 1)
  ((1.2))
  ((1.0))))))
```

In addition to the tree we need durations for each phone in the set

```
(set! spanish_el_phone_data
' (
  (# 0.0 0.250)
  (a 0.0 0.090)
  (e 0.0 0.090)
  (i 0.0 0.080)
  (o 0.0 0.090)
  (u 0.0 0.080)
  (b 0.0 0.065)
  (ch 0.0 0.135)
  (d 0.0 0.060)
  (f 0.0 0.100)
  (g 0.0 0.080)
  (j 0.0 0.100)
  (k 0.0 0.100)
  (l 0.0 0.080)
  (ll 0.0 0.105)
  (m 0.0 0.070)
  (n 0.0 0.080)
  (ny 0.0 0.110)
  (p 0.0 0.100)
  (r 0.0 0.030)
  (rr 0.0 0.080)
  (s 0.0 0.110)
  (t 0.0 0.085)
  (th 0.0 0.100)
  (x 0.0 0.130)
))
```

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

24.2.6 Waveform synthesis

There are a number of choices for waveform synthesis currently supported. MBROLA supports Spanish, so we could use that. But their Spanish diphones in fact use a slightly different phoneset so we would need to change the above definitions to use it effectively. Here we will use a diphone database for Spanish recorded by Eduardo Lopez when he was a Masters student some years ago.

Here we simply load our pre-built diphone database

```
(us_diphone_init
  (list
    '(name "el_lpc_group")
    (list 'index_file
      (path-append spanish_el_dir "group/ellpc11k.group"))
    '(grouped "true")
    '(default_diphone "#-#")))
```

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

24.2.7 Voice selection function

The standard way to define a voice in Festival is to define a function of the form `voice_NAME` which selects all the appropriate parameters. Because the definition below follows the above definitions we know that everything appropriate has been loaded into Festival and hence we just need to select the appropriate a parameters.

```
(define (voice_spanish_el)
  "(voice_spanish_el)
  Set up synthesis for Male Spanish speaker: Eduardo Lopez"
  (voice_reset)
  (Parameter.set 'Language 'spanish)
  ;; Phone set
  (Parameter.set 'PhoneSet 'spanish)
  (PhoneSet.select 'spanish)
  (set! pos_lex_name nil)
  ;; Phrase break prediction by punctuation
  (set! pos_supported nil)
  ;; Phrasing
  (set! phrase_cart_tree spanish_phrase_cart_tree)
  (Parameter.set 'Phrase_Method 'cart_tree)
  ;; Lexicon selection
  (lex.select "spanish")
  ;; Accent prediction
  (set! int_accent_cart_tree spanish_accent_cart_tree)
  (set! int_simple_params spanish_el_int_simple_params)
  (Parameter.set 'Int_Method 'Simple)
  ;; Duration prediction
  (set! duration_cart_tree spanish_dur_tree)
  (set! duration_ph_info spanish_el_phone_data)
  (Parameter.set 'Duration_Method 'Tree_ZScores)
  ;; Waveform synthesizer: diphones
  (Parameter.set 'Synth_Method 'UniSyn)
  (Parameter.set 'us_sigpr 'lpc)
  (us_db_select 'el_lpc_group)

  (set! current-voice 'spanish_el)
)

(provide 'spanish_el)
```

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

24.2.8 Last remarks

We save the above definitions in a file ``spanish_el.scm'`. Now we can declare the new voice to Festival. See section [24.3 Defining a new voice](#), for a description of methods for adding new voices. For testing purposes we can explicitly load the file ``spanish_el.scm'`

The voice is now available for use in festival.

```
festival> (voice_spanish_el)
spanish_el
festival> (SayText "hola amigos")
<Utterance 0x04666>
```

As you can see adding a new voice is not very difficult. Of course there is quite a lot more than the above to add a high quality robust voice to Festival. But as we can see many of the basic tools that we wish to use already exist. The main difference between the above voice and the English voices already in Festival are that their models are better trained from databases. This produces, in general, better results, but the concepts behind them are basically the same. All of those trainable methods may be parameterized with data for new voices.

As Festival develops, more modules will be added with better support for training new voices so in the end we hope that adding in high quality new voices is actually as simple as (or indeed simpler than) the above description.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

24.2.9 Resetting globals

Because the version of Scheme used in Festival only has a single flat name space it is unfortunately too easy for voices to set some global which accidentally affects all other voices selected after it. Because of this problem we have introduced a convention to try to minimise the possibility of this becoming a problem. Each voice function defined should always call `voice_reset` at the start. This will reset any globals and also call a tidy up function provided by the previous voice function.

Likewise in your new voice function you should provide a tidy up function to reset any non-standard global variables you set. The function `current_voice_reset` will be called by `voice_reset`. If the value of `current_voice_reset` is `nil` then it is not called. `voice_reset` sets `current_voice_reset` to `nil`, after calling it.

For example suppose some new voice requires the audio device to be directed to a different machine. In this example we make the giant's voice go through the netaudio machine `big_speakers` while the standard voice go through `small_speakers`.

Although we can easily select the machine `big_speakers` as out when our `voice_giant` is called, we also need to set it back when the next voice is selected, and don't want to have to modify every other voice defined in the system. Let us first define two functions to selection the audio output.

```
(define (select_big)
  (set! giant_previous_audio (getenv "AUDIOSERVER")))
  (setenv "AUDIOSERVER" "big_speakers"))

(define (select_normal)
  (setenv "AUDIOSERVER" giant_previous_audio))
```

Note we save the previous value of `AUDIOSERVER` rather than simply assuming it was `small_speakers`.

Our definition of `voice_giant` definition of `voice_giant` will look something like

```
(define (voice_giant)
  "comment comment ..."
  (voice_reset) ;; get into a known state
  (select_big)
  ;; other giant voice parameters
  ...

  (set! current_voice_rest select_normal)
  (set! current-voice 'giant))
```

The obvious question is which variables should a voice reset. Unfortunately there is not a definitive answer to that. To a certain extent I don't want to define that list as there will be many variables that will be added by various people in Festival which are not in the original distribution and we don't want to restrict them. The longer term answer is some form of partitioning of the Scheme name space perhaps having voice local variables (cf. Emacs buffer local variables). But ultimately a voice may set global variables which could redefine the operation of later selected voices and there seems no real way to stop that, and keep the generality of the system.

Note the convention of setting the global `current-voice` as the end of any voice definition file. We do not enforce this but probably should. The variable `current-voice` at any time should identify the current voice, the voice description information (described below) will relate this name to properties identifying it.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

24.3 Defining a new voice

As there are a number of voices available for Festival and they may or may not exist in different installations we have tried to make it as simple as possible to add new voices to the system without having to change any of the basic distribution. In fact if the voices use the following standard method for describing themselves it is merely a matter of unpacking them in order for them to be used by the system.

The variable `voice-path` contains a list of directories where voices will be automatically searched for. If this is not set it is set automatically by appending ``/voices/'` to all paths in `festival load-path`. You may add new directories explicitly to this variable in your ``sitevars.scm'` file or your own ``.festivalrc'` as you wish.

Each voice directory is assumed to be of the form

```
LANGUAGE/VOICENAME/
```

Within the `VOICENAME/` directory itself it is assumed there is a file ``festvox/VOICENAME.scm'` which when loaded will define the voice itself. The actual voice function should be called `voice_VOICENAME`.

For example the voices distributed with the standard Festival distribution all unpack in ``festival/lib/voices'`. The American voice ``ked_diphone'` unpacks into

```
festival/lib/voices/english/ked_diphone/
```

Its actual definition file is in

```
festival/lib/voices/english/ked_diphone/festvox/ked_diphone.scm
```

Note the name of the directory and the name of the Scheme definition file must be the same.

Alternative voices using perhaps a different encoding of the database but the same front end may be defined in the same way by using symbolic links in the language directory to the main directory. For example a PSOLA version of the `ked` voice may be defined in

```
festival/lib/voices/english/ked_diphone/festvox/ked_psola.scm
```

Adding a symbol link in `festival/lib/voices/english/' to `ked_diphone' called `ked_psola' will allow that voice to be automatically registered when Festival starts up.

Note that this method doesn't actually load the voices it finds, that could be prohibitively time consuming to the start up process. It blindly assumes that there is a file `VOICENAME/festvox/VOICENAME.scm' to load. An autoload definition is given for `voice_VOICENAME` which when called will load that file and call the real definition if it exists in the file.

This is only a recommended method to make adding new voices easier, it may be ignored if you wish. However we still recommend that even if you use your own conventions for adding new voices you consider the autoload function to define them in, for example, the `siteinit.scm' file or `.festivalrc'. The autoload function takes three arguments: a function name, a file containing the actual definition and a comment. For example a definition of voice can be done explicitly by

```
(autoload voice_f2b "/home/awb/data/f2b/ducs/f2b_ducs"
  "American English female f2b"))
```

Of course you can also load the definition file explicitly if you wish.

In order to allow the system to start making intelligent use of voices we recommend that all voice definitions include a call to the function `voice_proclaim` this allows the system to know some properties about the voice such as language, gender and dialect. The `proclaim_voice` function takes two arguments a name (e.g. `rab_diphone`) and an assoc list of features and names. Currently we require language, gender, dialect and description. The last being a textual description of the voice itself. An example proclamation is

```
(proclaim_voice
 'rab_diphone
 '((language english)
 (gender male)
 (dialect british)
 (description
 "This voice provides a British RP English male voice using a
 residual excited LPC diphone synthesis method. It uses a
 modified Oxford Advanced Learners' Dictionary for pronunciations.
 Prosodic phrasing is provided by a statistically trained model
 using part of speech and local distribution of breaks. Intonation
 is provided by a CART tree predicting ToBI accents and an F0
 contour generated from a model trained from natural speech. The
 duration model is also trained from data using a CART tree.")))
```

There are functions to access a description. `voice.description` will return the description for a given voice and will load that voice if it is not already loaded. `voice.describe` will describe the given voice by synthesizing the textual description using the current voice. It would be nice to use the voice itself to give a self introduction but unfortunately that introduces a problem of deciding which language the description should be in, we are not all as fluent in Welsh as we'd like to be.

The function `voice.list` will list the *potential* voices in the system. These are the names of voices which have been found in the `voice-path`. As they have not actually been loaded they can't actually be confirmed as usable voices. One solution to this would be to load all voices at start up time which would allow confirmation they exist and to get their full description through `proclaim_voice`. But start up is already too slow in Festival so we have to accept this state for the time being. Splitting the description of the voice from the actual definition is a possible solution to this problem but we have not yet looked into this.

25. Tools

A number of basic data manipulation tools are supported by Festival. These often make building new modules very easy and are already used in many of the existing modules. They typically offer a Scheme method for entering data, and Scheme and C++ functions for evaluating it.

[25.1 Regular expressions](#)

[25.2 CART trees](#) Building and using CART

[25.3 Ngrams](#) Building and using Ngrams

[25.4 Viterbi decoder](#) Using the Viterbi decoder

[25.5 Linear regression](#) Building and using linear regression models

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

25.1 Regular expressions

Regular expressions are a formal method for describing a certain class of mathematical languages. They may be viewed as patterns which match some set of strings. They are very common in many software tools such as scripting languages like the UNIX shell, PERL, awk, Emacs etc. Unfortunately the exact form of regular expressions often differs slightly between different applications making their use often a little tricky.

Festival support regular expressions based mainly of the form used in the GNU libg++ `Regex` class, though we have our own implementation of it. Our implementation (`EST_Regex`) is actually based on Henry Spencer's ``regex.c'` as distributed with BSD 4.4.

Regular expressions are represented as character strings which are interpreted as regular expressions by certain Scheme and C++ functions. Most characters in a regular expression are treated as literals and match only that character but a number of others have special meaning. Some characters may be escaped with preceding backslashes to change them from operators to literals (or sometime literals to operators).

`.` Matches any character.
`$` matches end of string
`^` matches beginning of string
`X*` matches zero or more occurrences of X, X may be a character, range of parenthesized expression.
`X+` matches one or more occurrences of X, X may be a character, range of parenthesized expression.
`X?` matches zero or one occurrence of X, X may be a character, range of parenthesized expression.
`[. . .]` a ranges matches an of the values in the brackets. The range operator "-" allows specification of ranges e.g. a-z for all lower case characters. If the first character of the range is ^ then it matches anything character except those specified in the range. If you wish - to be in the range you must put that first.
`\\(. . . \\)` Treat contents of parentheses as single object allowing operators *, +, ? etc to operate on more than single characters.
`X\\|Y` matches either X or Y. X or Y may be single characters, ranges or parenthesized expressions.

Note that actually only one backslash is needed before a character to escape it but because these expressions are most often contained with Scheme or C++ strings, the escape mechanism for those strings requires that backslash itself be escaped, hence you will most often be required to type two backslashes.

Some examples may help in understanding the use of regular expressions.

`a.b`
matches any three letter string starting with an a and ending with a b.

`.*a`
matches any string ending in an a

`.*a.*`
matches any string containing an a

`[A-Z].*`
matches any string starting with a capital letter

`[0-9]+`
matches any string of digits

`-?[0-9]+\.\([0-9]+\)`
matches any positive or negative real number. Note the optional preceding minus sign and the optional part contain the point and following numbers. The point itself must be escaped as dot on its own matches any character.

`[^aeiouAEIOU]+`
matches any non-empty string which doesn't contain a vowel

`\\([Ss]at\\(urday\\)\)\)?\\|\\([Ss]un\\(day\\)\)\)`
matches Saturday and Sunday in various ways

The Scheme function `string-matches` takes a string and a regular expression and returns `t` if the regular expression matches the string and `nil` otherwise.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

25.2 CART trees

One of the basic tools available with Festival is a system for building and using Classification and Regression Trees (*breiman84*). This standard statistical method can be used to predict both categorical and continuous data from a set of feature vectors.

The tree itself contains yes/no questions about features and ultimately provides either a probability distribution, when predicting categorical values (classification tree), or a mean and standard deviation when predicting continuous values (regression tree). Well defined techniques can be used to construct an optimal tree from a set of training data. The program, developed in conjunction with Festival, called `wagon`, distributed with the speech tools, provides a basic but ever increasingly powerful method for constructing trees.

A tree need not be automatically constructed, CART trees have the advantage over some other automatic training methods, such as neural networks and linear regression, in that their output is more readable and often understandable by humans. Importantly this makes it possible to modify them. CART trees may also be fully hand constructed. This is used, for example, in generating some duration models for languages we do not yet have full databases to train from.

A CART tree has the following syntax

```

CART ::= QUESTION-NODE || ANSWER-NODE
QUESTION-NODE ::= ( QUESTION YES-NODE NO-NODE )
YES-NODE ::= CART
NO-NODE ::= CART
QUESTION ::= ( FEATURE in LIST )
QUESTION ::= ( FEATURE is STRVALUE )
QUESTION ::= ( FEATURE = NUMVALUE )
QUESTION ::= ( FEATURE > NUMVALUE )
QUESTION ::= ( FEATURE < NUMVALUE )
QUESTION ::= ( FEATURE matches REGEX )
ANSWER-NODE ::= CLASS-ANSWER || REGRESS-ANSWER
CLASS-ANSWER ::= ( (VALUE0 PROB) (VALUE1 PROB) ... MOST-PROB-VALUE )
REGRESS-ANSWER ::= ( ( STANDARD-DEVIATION MEAN ) )

```

Note that answer nodes are distinguished by their car not being atomic.

The interpretation of a tree is with respect to a `Stream_Item` The *FEATURE* in a tree is a standard feature (see section [14.6 Features](#)).

The following example tree is used in one of the Spanish voices to predict variations from average durations.

```

(set! spanish_dur_tree
  ,
  (set! spanish_dur_tree
    ,
    ((R:SylStructure.parent.R:Syllable.p.syl_break > 1) ;; clause initial
     ((R:SylStructure.parent.stress is 1)
      ((1.5))
      ((1.2))))
     ((R:SylStructure.parent.syl_break > 1) ;; clause final
      ((R:SylStructure.parent.stress is 1)
       ((2.0))
       ((1.5)))
      ((R:SylStructure.parent.stress is 1)
       ((1.2))
       ((1.0))))))

```

It is applied to the segment stream to give a factor to multiply the average by.

wagon is constantly improving and with version 1.2 of the speech tools may now be considered fairly stable for its basic operations. Experimental features are described in help it gives. See the Speech Tools manual for a more comprehensive discussion of using `wagon`.

However the above format of trees is similar to those produced by many other systems and hence it is reasonable to translate their formats into one which Festival can use.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

25.3 Ngrams

Bigram, trigrams, and general ngrams are used in the part of speech tagger and the phrase break predictor. An Ngram C++ Class is defined in the speech tools library and some simple facilities are added within Festival itself.

Ngrams may be built from files of tokens using the program `ngram_build` which is part of the speech tools. See the speech tools documentation for details.

Within Festival ngrams may be named and loaded from files and used when required. The LISP function `load_ngram` takes a name and a filename as argument and loads the Ngram from that file. For an example of its use once loaded see ``src/modules/base/pos.cc'` or ``src/modules/base/phrasify.cc'`.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

25.4 Viterbi decoder

Another common tool is a Viterbi decoder. This C++ Class is defined in the speech tools library ``speech_tools/include/EST_viterbi.h'` and ``speech_tools/stats/EST_viterbi.cc'`. A Viterbi decoder requires two functions at declaration time. The first constructs candidates at each stage, while the second combines paths. A number of options are available (which may change).

The prototypical example of use is in the part of speech tagger which using standard Ngram models to predict probabilities of tags. See ``src/modules/base/pos.cc'` for an example.

The Viterbi decoder can also be used through the Scheme function `Gen_Viterbi`. This function respects the parameters defined in the variable `get_vit_params`. Like other modules this parameter list is an assoc list of feature name and value. The parameters supported are:

`Relation`

The name of the relation the decoder is to be applied to.

`cand_function`

A function that is to be called for each item that will return a list of candidates (with probabilities).

`return_feat`

The name of a feature that the best candidate is to be returned in for each item in the named relation.

`p_word`

The previous word to the first item in the named relation (only used when ngrams are the "language model").

`pp_word`

The previous previous word to the first item in the named relation (only used when ngrams are the "language model").

`ngramname`

the name of an ngram (loaded by `ngram.load`) to be used as a "language model".

`wfstname`

the name of a WFST (loaded by `wfst.load`) to be used as a "language model", this is ignored if an `ngramname` is also specified.

`debug`

If specified more debug features are added to the items in the relation.

`gscale_p`

Grammar scaling factor.

Here is a short example to help make the use of this facility clearer.

There are two parts required for the Viterbi decode a set of candidate observations and some "language model". For the math to work properly the candidate observations must be reverse probabilities (for each candidate as given what is the probability of the observation, rather than the probability of the candidate given the observation). These can be calculated for the probabilities candidate given the observation divided by the probability of the candidate in isolation.

For the sake of simplicity let us assume we have a lexicon of words to distribution of part of speech tags with reverse probabilities. And an tri-gram called `pos-tri-gram` over ngram sequences of part of speech tags. First we must define the candidate function

```
(define (pos_cand_function w)
  ;; select the appropriate lexicon
  (lex.select 'pos_lex)
  ;; return the list of cand_s with rprobs
  (cadr
   (lex.lookup (item.name w) nil)))
```

The returned candidate list would look something like

```
( ( jj -9.872) ( vbd -6.284) ( vbn -5.565) )
```

Our part of speech tagger function would look something like this

```
(define (pos_tagger utt)
  (set! get_vit_params
    (list
     (list 'Relation "Word")
     (list 'return_feat 'pos_tag)
     (list 'p_word "punc")
     (list 'pp_word "nn")
     (list 'ngramname "pos-tri-gram")
     (list 'cand_function 'pos_cand_function)))
  (Gen_Viterbi utt)
  utt)
```

this will assign the optimal part of speech tags to each word in utt.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

25.5 Linear regression

The linear regression model takes models built from some external package and finds coefficients based on the features and weights. A model consists of a list of features. The first should be the atom `Intercept` plus a value. The following in the list should consist of a feature (see section [14.6 Features](#)) followed by a weight. An optional third element may be a list of atomic values. If the result of the feature is a member of this list the feature's value is treated as 1 else it is 0. This third argument allows an efficient way to map categorical values into numeric values. For example, from the F0 prediction model in ``lib/f2bf0lr.scm'`. The first few parameters are

```
(set! f2b_f0_lr_start
 '(
  ( Intercept 160.584956 )
  ( Word.Token.EMPH 36.0 )
  ( pp.tobi_accent 10.081770 (H*) )
  ( pp.tobi_accent 3.358613 (!H*) )
  ( pp.tobi_accent 4.144342 (*? X*? H*!H* * L+H* L+!H*) )
  ( pp.tobi_accent -1.111794 (L*) )
  ...
 )
```

Note the feature `pp.tobi_accent` returns an atom, and is hence tested with the map groups specified as third arguments.

Models may be built from feature data (in the same format as ``wagon'` using the ``ols'` program distributed with the speech tools library.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

26. Building models from databases

Because our research interests tend towards creating statistical models trained from real speech data, Festival offers various support for extracting information from speech databases, in a way suitable for building models.

Models for accent prediction, F0 generation, duration, vowel reduction, homograph disambiguation, phrase break assignment and unit selection have been built using Festival to extract and process various databases.

- [26.1 Labelling databases](#) Phones, syllables, words etc.
- [26.2 Extracting features](#) Extraction of model parameters.
- [26.3 Building models](#) Building stochastic models from features

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

26.1 Labelling databases

In order for Festival to use a database it is most useful to build utterance structures for each utterance in the database. As discussed earlier, utterance structures contain relations of items. Given such a structure for each utterance in a database we can easily read in the utterance representation and access it, dumping information in a normalised way allowing for easy building and testing of models.

Of course the level of labelling that exists, or that you are willing to do by hand or using some automatic tool, for a particular database will vary. For many purposes you will at least need phonetic labelling. Hand labelled data is still better than auto-labelled data, but that could change. The size and consistency of the data is important too.

For this discussion we will assume labels for: segments, syllables, words, phrases, intonation events, pitch targets. Some of these can be derived, some need to be labelled. This would not fail with less labelling but of course you wouldn't be able to extract as much information from the result.

In our databases these labels are in Entropic's Xlabel format, though it is fairly easy to convert any reasonable format.

Segment

These give phoneme labels for files. Note the these labels *must* be members of the phoneset that you will be using for this database. Often phone label files may contain extra labels (e.g. beginning and end silence) which are not really part of the phoneset. You should remove (or re-label) these phones accordingly.

Word

Again these will need to be provided. The end of the word should come at the last phone in the word (or just after). Pauses/silences should not be part of the word.

Syllable

There is a chance these can be automatically generated from Word and Segment files given a lexicon. Ideally these should include lexical stress.

IntEvent

These should ideally mark accent/boundary tone type for each syllable, but this almost definitely requires hand-labelling. Also given that hand-labelling of accent type is harder and not as accurate, it is arguable that anything other than accented vs. non-accented can be used reliably.

Phrase

This could just mark the last non-silence phone in each utterance, or before any silence phones in the whole utterance.

Target

This can be automatically derived from an F0 file and the Segment files. A marking of the mean F0 in each voiced phone seem to give adequate results.

Once these files are created an utterance file can be automatically created from the above data. Note it is pretty easy to get the streams right but getting the relations between the streams is much harder. Firstly labelling is rarely accurate and small windows of error must be allowed to ensure things line up properly. The second problem is that

some label files identify point type information (IntEvent and Target) while others identify segments (e.g. Segment, Words etc.). Relations have to know this in order to get it right. For example is not right for all syllables between two IntEvents to be linked to the IntEvent, only to the Syllable the IntEvent is within.

The script ``festival/examples/make_utts'` is an example Festival script which automatically builds the utterance files from the above labelled files.

The script, by default assumes, a hierarchy in an database directory of the following form. Under a directory ``festival/`` where all festival specific database information can be kept, a directory ``relations/`` contains a subdirectory for each basic relation (e.g. ``Segment/``, ``Syllable/``, etc.) Each of which contains the basic label files for that relation.

The following command will build a set of utterance structures (including building the relations that link between these basic relations).

```
make_utts -phoneset radio festival/relation/Segment/*.Segment
```

This will create utterances in ``festival/utts/``. There are a number of options to ``make_utts'` use ``-h'` to find them. The ``-eval'` option allows extra scheme code to be loaded which may be called by the utterance building process. The function `make_utts_user_function` will be called on all utterance created. Redefining that in database specific loaded code will allow database specific fixed to the utterance.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

26.2 Extracting features

The easiest way to extract features from a labelled database of the form described in the previous section is by loading in each of the utterance structures and dumping the desired features.

Using the same mechanism to extract the features as will eventually be used by models built from the features has the important advantage of avoiding spurious errors easily introduced when collecting data. For example a feature such as `n.accent` in a Festival utterance will be defined as 0 when there is no next accent. Extracting all the accents and using an external program to calculate the next accent may make a different decision so that when the generated model is used a different value for this feature will be produced. Such mismatches in training models and actual use are unfortunately common, so using the same mechanism to extract data for training, and for actual use is worthwhile.

The recommended method for extracting features is using the festival script ``dumpfeats'`. It basically takes a list of feature names and a list of utterance files and dumps the desired features.

Features may be dumped into a single file or into separate files one for each utterance. Feature names may be specified on the command line or in a separate file. Extra code to define new features may be loaded too.

For example suppose we wanted to save the features for a set of utterances include the duration, phone name, previous and next phone names for all segments in each utterance.

```
dumpfeats -feats "(segment_duration name p.name n.name)" \  
-output feats/%s.dur -relation Segment \  
festival/utts/*.utt
```

This will save these features in files named for the utterances they come from in the directory ``feats/``. The argument to ``-feats'` is treated as literal list only if it starts with a left parenthesis, otherwise it is treated as a filename contain named features (unbracketed).

Extra code (for new feature definitions) may be loaded through the ``-eval'` option. If the argument to ``-eval'` starts with a left parenthesis it is treated as an s-expression rather than a filename and is evaluated. If argument ``-`

output ' contains "%s" it will be filled in with the utterance's filename, if it is a simple filename the features from all utterances will be saved in that same file. The features for each item in the named relation are saved on a single line.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

26.3 Building models

This section describes how to build models from data extracted from databases as described in the previous section. It uses the CART building program, 'wagon' which is available in the speech tools distribution. But the data is suitable for many other types of model building techniques, such as linear regression or neural networks.

Wagon is described in the speech tools manual, though we will cover simple use here. To use Wagon you need a datafile and a data description file.

A datafile consists of a number of vectors one per line each containing the same number of fields. This, not coincidentally, is exactly the format produced by 'dumpfeats' described in the previous section. The data description file describes the fields in the datafile and their range. Fields may be of any of the following types: class (a list of symbols), floats, or ignored. Wagon will build a classification tree if the first field (the predictee) is of type class, or a regression tree if the first field is a float. An example data description file would be

```
(
( duration float )
( name # @ @@ a aa ai au b ch d dh e e@ ei f g h i i@ ii jh k l m n
  ng o oi oo ou p r s sh t th u u@ uh uu v w y z zh )
( n.name # @ @@ a aa ai au b ch d dh e e@ ei f g h i i@ ii jh k l m n
  ng o oi oo ou p r s sh t th u u@ uh uu v w y z zh )
( p.name # @ @@ a aa ai au b ch d dh e e@ ei f g h i i@ ii jh k l m n
  ng o oi oo ou p r s sh t th u u@ uh uu v w y z zh )
( R:SylStructure.parent.position_type 0 final initial mid single )
( pos_in_syl float )
( syl_initial 0 1 )
( syl_final 0 1 )
( R:SylStructure.parent.R:Syllable.p.syl_break 0 1 3 )
( R:SylStructure.parent.syl_break 0 1 3 4 )
( R:SylStructure.parent.R:Syllable.n.syl_break 0 1 3 4 )
( R:SylStructure.parent.R:Syllable.p.stress 0 1 )
( R:SylStructure.parent.stress 0 1 )
( R:SylStructure.parent.R:Syllable.n.stress 0 1 )
)
```

The script 'speech_tools/bin/make_wagon_desc' goes some way to helping. Given a datafile and a file containing the field names, it will construct an approximation of the description file. This file should still be edited as all fields are treated as of type class by 'make_wagon_desc' and you may want to change them some of them to float.

The data file must be a single file, although we created a number of feature files by the process described in the previous section. From a list of file ids select, say, 80% of them, as training data and cat them into a single datafile. The remaining 20% may be catted together as test data.

To build a tree use a command like

```
wagon -desc DESCFILE -data TRAINFILE -test TESTFILE
```

The minimum cluster size (default 50) may be reduced using the command line option -stop plus a number.

Varying the features and stop size may improve the results.

Building the models and getting good figures is only one part of the process. You must integrate this model into Festival if its going to be of any use. In the case of CART trees generated by Wagon, Festival supports these directly. In the case of CART trees predicting zscores, or factors to modify duration averages, ees can be used as is.

Note there are other options to Wagon which may help build better CART models. Consult the chapter in the speech tools manual on Wagon for more information.

Other parts of the distributed system use CART trees, and linear regression models that were training using the processes described in this chapter. Some other parts of the distributed system use CART trees which were written by hand and may be improved by properly applying these processes.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

27. Programming

This chapter covers aspects of programming within the Festival environment, creating new modules, and modifying existing ones. It describes basic Classes available and gives some particular examples of things you may wish to add.

[27.1 The source code](#) A walkthrough of the source code

[27.2 Writing a new module](#) Example access of an utterance

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

27.1 The source code

The ultimate authority on what happens in the system lies in the source code itself. No matter how hard we try, and how automatic we make it, the source code will always be ahead of the documentation. Thus if you are going to be using Festival in a serious way, familiarity with the source is essential.

The lowest level functions are catered for in the Edinburgh Speech Tools, a separate library distributed with Festival. The Edinburgh Speech Tool Library offers the basic utterance structure, waveform file access, and other various useful low-level functions which we share between different speech systems in our work. See section 'Overview' in *Edinburgh Speech Tools Library Manual*.

The directory structure for the Festival distribution reflects the conceptual split in the code.

`^ ./bin/ '`

The user-level executable binaries and scripts that are part of the festival system. These are simple symbolic links to the binaries or if the system is compiled with shared libraries small wrap-around shell scripts that set LD_LIBRARY_PATH appropriately

`^ ./doc/ '`

This contains the texinfo documentation for the whole system. The `Makefile` constructs the info and/or html version as desired. Note that the `festival` binary itself is used to generate the lists of functions and variables used within the system, so must be compiled and in place to generate a new version of the documentation.

`^ ./examples/ '`

This contains various examples. Some are explained within this manual, others are there just as examples.

`^ ./lib/ '`

The basic Scheme parts of the system, including `init.scm` the first file loaded by `festival` at start-up time. Depending on your installation, this directory may also contain subdirectories containing lexicons, voices and databases. This directory and its sub-directories are used by Festival at run-time.

`^ ./lib/etc/ '`

Executables for Festival's internal use. A subdirectory containing at least the audio spooler will be

automatically created (one for each different architecture the system is compiled on). Scripts are added to this top level directory itself.

`. /lib/voices/ '

By default this contains the voices used by Festival including their basic Scheme set up functions as well as the diphone databases.

`. /lib/dicts/ '

This contains various lexicon files distributed as part of the system.

`. /config/ '

This contains the basic `Makefile` configuration files for compiling the system (run-time configuration is handled by Scheme in the `lib/` directory). The file `config/config` created as a copy of the standard `config/config-dist` is the installation specific configuration. In most cases a simple copy of the distribution file will be sufficient.

`. /src/ '

The main C++/C source for the system.

`. /src/lib/ '

Where the `libFestival.a` is built.

`. /src/include/ '

Where include files shared between various parts of the system live. The file `festival.h` provides access to most of the parts of the system.

`. /src/main/ '

Contains the top level C++ files for the actual executables. This is directory where the executable binary `festival` is created.

`. /src/arch/ '

The main core of the Festival system. At present everything is held in a single sub-directory `./src/arch/festival/`. This contains the basic core of the synthesis system itself. This directory contains lisp front ends to access the core utterance architecture, and phonesets, basic tools like, client/server support, ngram support, etc, and an audio spooler.

`. /src/modules/ '

In contrast to the `arch/` directory this contains the non-core parts of the system. A set of basic example modules are included with the standard distribution. These are the parts that do the synthesis, the other parts are just there to make module writing easier.

`. /src/modules/base/ '

This contains some basic simple modules that weren't quite big enough to deserve their own directory. Most importantly it includes the `Initialize` module called by many synthesis methods which sets up an utterance structure and loads in initial values. This directory also contains phrasing, part of speech, and word (syllable and phone construction from words) modules.

`. /src/modules/Lexicon/ '

This is not really a module in the true sense (the `Word` module is the main user of this). This contains functions to construct, compile, and access lexicons (entries of words, part of speech and pronunciations). This also contains a letter-to-sound rule system.

`. /src/modules/Intonation/ '

This contains various intonation systems, from the very simple to quite complex parameter driven intonation systems.

`. /src/modules/Duration/ '

This contains various duration prediction systems, from the very simple (fixed duration) to quite complex parameter driven duration systems.

`. /src/modules/UniSyn/ '

A basic diphone synthesizer system, supporting a simple database format (which can be grouped into a more efficient binary representation). It is multi-lingual, and allows multiple databases to be loaded at once. It offers a choice of concatenation methods for diphones: residual excited LPC or PSOLA (TM) (which is not distributed)

`. /src/modules/Text/ '

Various text analysis functions, particularly the tokenizer and utterance segmenter (from arbitrary files). This directory also contains the support for text modes and SGML.

`. /src/modules/donovan/ '

An LPC based diphone synthesizer. Very small and neat.

`. /src/modules/rxp/ '

The Festival/Scheme front end to An XML parser written by Richard Tobin from University of Edinburgh's Language Technology Group.. rxp is now part of the speech tools rather than just Festival.

`. /src/modules/parser/ '

A simple interface to the Stochastic Context Free Grammar parser in the speech tools library.

`./src/modules/diphone`

An optional module contain the previously used diphone synthesizer.

`./src/modules/clunits`

A partial implementation of a cluster unit selection algorithm as described in *black97c*.

`./src/modules/Database rjc_synthesis`

This consist of a new set of modules for doing waveform synthesis. They are intended to unit size independent (e.g. diphone, phone, non-uniform unit). Also selection, prosodic modification, joining and signal processing are separately defined. Unfortunately this code has not really been exercised enough to be considered stable to be used in the default synthesis method, but those working on new synthesis techniques may be interested in integration using these new modules. They may be updated before the next full release of Festival.

`./src/modules/*`

Other optional directories may be contained here containing various research modules not yet part of the standard distribution. See below for descriptions of how to add modules to the basic system.

One intended use of Festival is offer a software system where new modules may be easily tested in a stable environment. We have tried to make the addition of new modules easy, without requiring complex modifications to the rest of the system.

All of the basic modules should really be considered merely as example modules. Without much effort all of them could be improved.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

27.2 Writing a new module

This section gives a simple example of writing a new module. showing the basic steps that must be done to create and add a new module that is available for the rest of the system to use. Note many things can be done solely in Scheme now and really only low-level very intensive things (like waveform synthesizers) need be coded in C++.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

27.2.1 Example 1: adding new modules

The example here is a duration module which sets durations of phones for a given list of averages. To make this example more interesting, all durations in accented syllables are increased by 1.5. Note that this is just an example for the sake of one, this (and much better techniques) could easily done within the system as it is at present using a hand-crafted CART tree.

Our new module, called `Duration_Simple` can most easily be added to the ``./src/Duration/`` directory in a file ``simdur.cc``. You can worry about the copyright notice, but after that you'll probably need the following includes

```
#include <festival.h>
```

The module itself must be declared in a fixed form. That is receiving a single LISP form (an utterance) as an argument and returning that LISP form at the end. Thus our definition will start

```
LISP FT_Duration_Simple(LISP utt)
{
```

Next we need to declare an utterance structure and extract it from the LISP form. We also make a few other variable declarations


```

EST_Utterance *u = get_c_utt(utt);
EST_Item *s;
float end=0.0, dur;
LISP ph_avgs, ldur;

```

We cannot list the average durations for each phone in the source code as we cannot tell which phoneset we are using (or what modifications we want to make to durations between speakers). Therefore the phone and average duration information is held in a Scheme variable for easy setting at run time. To use the information in our C++ domain we must get that value from the Scheme domain. This is done with the following statement.

```

ph_avgs = siod_get_lval("phoneme_averages", "no phoneme durations");

```

The first argument to `siod_get_lval` is the Scheme name of a variable which has been set to an assoc list of phone and average duration before this module is called. See the variable `phone_durations` in ``lib/mrpa_durs.scm'` for the format. The second argument to `siod_get_lval` is an error message to be printed if the variable `phoneme_averages` is not set. If the second argument to `siod_get_lval` is `NULL` then no error is given and if the variable is unset this function simply returns the Scheme value `nil`.

Now that we have the duration data we can go through each segment in the utterance and add the duration. The loop looks like

```

for (s=u->relation("Segment")->head(); s != 0; s = next(s))
{

```

We can lookup the average duration of the current segment name using the function `siod_assoc_str`. As arguments, it takes the segment name `s->name()` and the assoc list of phones and duration.

```

ldur = siod_assoc_str(s->name(), ph_avgs);

```

Note the return value is actually a LISP pair (phone name and duration), or `nil` if the phone isn't in the list. Here we check if the segment is in the list. If it is not we print an error and set the duration to 100 ms, if it is in the list the floating point number is extracted from the LISP pair.

```

if (ldur == NIL)
{
    cerr << "Phoneme: " << s->name() << " no duration "
        << endl;
    dur = 0.100;
}
else
    dur = get_c_float(car(cdr(ldur)));

```

If this phone is in an accented syllable we wish to increase its duration by a factor of 1.5. To find out if it is accented we use the feature system to find the syllable this phone is part of and find out if that syllable is accented.

```

if (ffeature(s, "R:SylStructure.parent.accented") == 1)
    dur *= 1.5;

```

Now that we have the desired duration we increment the end duration with our predicted duration for this segment and set the end of the current segment.

```

    end += dur;
    s->fset("end", end);
}

```

Finally we return the utterance from the function.

```

    return utt;
}

```

Once a module is defined it must be declared to the system so it may be called. To do this one must call the function `festival_def_utt_module` which takes a LISP name, the C++ function name and a documentation string describing what the module does. This will automatically be available at run-time and added to the manual. The call

to this function should be added to the initialization function in the directory you are adding the module too. The function is called `festival_DIRNAME_init()`. If one doesn't exist you'll need to create it.

In `./src/Duration/` the function `festival_Duration_init()` is at the end of the file `dur_aux.cc`. Thus we can add our new modules declaration at the end of that function. But first we must declare the C++ function in that file. Thus above that function we would add

```
LISP FT_Duration_Simple(LISP args);
```

While at the end of the function `festival_Duration_init()` we would add

```
    festival_def_utt_module("Duration_Simple", FT_Duration_Simple,
    "(Duration_Simple UTT)\n\
    Label all segments with average duration ... ");
```

In order for our new file to be compiled we must add it to the `Makefile` in that directory, to the `SRCS` variable. Then when we type `make` in `./src/` our new module will be properly linked in and available for use.

Of course we are not quite finished. We still have to say when our new duration module should be called. When we set

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

for a voice it will use our new module, calls to the function `utt.synth` will use our new duration module.

Note in earlier versions of Festival it was necessary to modify the duration calling function in `lib/duration.scm` but that is no longer necessary.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

27.2.2 Example 2: accessing the utterance

In this example we will make more direct use of the utterance structure, showing the gory details of following relations in an utterance. This time we will create a module that will name all syllables with a concatenation of the names of the segments they are related to.

As before we need the same standard includes

```
#include "festival.h"
```

Now the definition the function

```
LISP FT_Name_Syls(LISP utt)
{
```

As with the previous example we are called with an utterance LISP object and will return the same. The first task is to extract the utterance object from the LISP object.

```
    EST_Utterance *u = get_c_utt(utt);
    EST_Item *syl, *seg;
```

Now for each syllable in the utterance we want to find which segments are related to it.

```
    for (syl=u->relation("Syllable")->head(); syl != 0; syl = next(syl))
    {
```

Here we declare a variable to cummulate the names of the segments.

```
EST_String sylname = "";
```

Now we iterate through the `SylStructure` daughters of the syllable. These will be the segments in that syllable.

```
for (seg=daughter1(syl,"SylStructure"); seg; seg=next(seg))
    sylname += seg->name();
```

Finally we set the syllables name to the concatenative name, and loop to the next syllable.

```
    syl->set_name(sylname);
}
```

Finally we return the LISP form of the utterance.

```
    return utt;
}
```

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

27.2.3 Example 3: adding new directories

In this example we will add a whole new subsystem. This will often be a common way for people to use Festival. For example let us assume we wish to add a formant waveform synthesizer (e.g like that in the free ``rsynth'` program). In this case we will add a whole new sub-directory to the modules directory. Let us call it ``rsynth/'`.

In the directory we need a ``Makefile'` of the standard form so we should copy one from one of the other directories, e.g. ``Intonation/'`. Standard methods are used to identify the source code files in a ``Makefile'` so that the ``.o'` files are properly added to the library. Following the other examples will ensure your code is integrated properly.

We'll just skip over the bit where you extract the information from the utterance structure and synthesize the waveform (see ``donovan/donovan.cc'` or ``diphone/diphone.cc'` for examples).

To get Festival to use your new module you must tell it to compile the directory's contents. This is done in ``festival/config/config'`. Add the line

```
ALSO_INCLUDE += rsynth
```

to the end of that file (there are simialr ones mentioned). Simply adding the name of the directory here will add that as a new module and the directory will be compiled.

What you must provide in your code is a function `festival_DIRNAME_init()` which will be called at initialization time. In this function you should call any further initialization require and define and new Lisp functions you to made available to the rest of the system. For example in the ``rsynth'` case we would define in some file in ``rsynth/'`

```

#include "festival.h"

static LISP utt_rtsynth(LISP utt)
{
    EST_Utterance *u = get_c_utt(utt);
    // Do format synthesis
    return utt;
}

void festival_rsynth_init()
{
    proclaim_module("rsynth");

    festival_def_utt_module("Rsynth_Synth", utt_rsynth,
        "(Rsynth_Synth UTT)
        A simple formant synthesizer");

    ...
}

```

Integration of the code in optional (and standard directories) is done by automatically creating ``src/modules/init_modules.cc'` for the list of standard directories plus those defined as `ALSO_INCLUDE`. A call to a function called `festival_DIRNAME_init()` will be made.

This mechanism is specifically designed so you can add modules to the system without changing anything in the standard distribution.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#)
[\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

27.2.4 Example 4: adding new LISP objects

This third examples shows you how to add a new Object to Scheme and add wraparounds to allow manipulation within the the Scheme (and C++) domain.

Like example 2 we are assuming this is done in a new directory. Suppose you have a new object called `Widget` that can transduce a string into some other string (with some optional continuous parameter. Thus, here we create a new file ``widget.cc'` like this

```

#include "festival.h"
#include "widget.h" // definitions for the widget class

```

In order to register the widgets as Lisp objects we actually need to register them as `EST_Val`'s as well. Thus we now need

```

VAL_REGISTER_CLASS(widget, Widget)
SIOD_REGISTER_CLASS(widget, Widget)

```

The first names given to these functions should be a short mnemonic name for the object that will be used in the defining of a set of access and construction functions. It of course must be unique within the whole systems. The second name is the name of the object itself.

To understand its usage we can add a few simple widget manipulation functions

```

LISP widget_load(LISP filename)
{
    EST_String fname = get_c_string(filename);
    Widget *w = new Widget;    // build a new widget

    if (w->load(fname) == 0)    // successful load
        return siod(w);
    else
    {
        cerr << "widget load: failed to load \"" << fname << "\" << endl;
        festival_error();
    }
    return NIL;    // for compilers that get confused
}

```

Note that the function `siod` constructs a LISP object from a widget, the class register macro defines that for you. Also note that when giving an object to a LISP object it then owns the object and is responsible for deleting it when garbage collection occurs on that LISP object. Care should be taken that you don't put the same object within different LISP objects. The macros `VAL_REGISTER_CLASS_NODEL` should be called if you do not want your give object to be deleted by the LISP system (this may cause leaks).

If you want refer to these functions in other files within your models you can use

```

VAL_REGISTER_CLASS_DCLS(widget,Widget)
SIOD_REGISTER_CLASS_DCLS(widget,Widget)

```

in a common `.h` file

The following defines a function that takes a LISP object containing a widget, applies some method and returns a string.

```

LISP widget_apply(LISP lwidget, LISP string, LISP param)
{
    Widget *w = widget(lwidget);
    EST_String s = get_c_string(string);
    float p = get_c_float(param);
    EST_String answer;

    answer = w->apply(s,p);

    return strintern(answer);
}

```

The function `widget`, defined by the registration macros, takes a LISP object and returns a pointer to the widget inside it. If the LISP object does not contain a widget an error will be thrown.

Finally you wish to add these functions to the Lisp system

```

void festival_widget_init()
{
    init_subr_1("widget.load", widget_load,
        "(widget.load FILENAME)\n\
Load in widget from FILENAME.");
    init_subr_3("widget.apply", widget_apply,
        "(widget.apply WIDGET INPUT VAL)\n\
Returns widget applied to string INPUT with float VAL.");
}

```

In your `Makefile` for this directory you'll need to add the include directory where `widget.h` is, if it is not contained within the directory itself. This is done through the make variable `LOCAL_INCLUDES` as

```
LOCAL_INCLUDES = -I/usr/local/widget/include
```

And for the linker you'll need to identify where your widget library is. In your `festival/config/config` file at the end add

```
COMPILERLIBS += -L/usr/local/widget/lib -lwidget
```

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

28. API

If you wish to use Festival within some other application there are a number of possible interfaces.

28.1 Scheme API	Programming in Scheme
28.2 Shell API	From Unix shell
28.3 Server/client API	Festival as a speech synthesis server
28.4 C/C++ API	Through function calls from C++.
28.5 C only API	Small independent C client access
28.6 Java and JSAPI	Synthesizing from Java

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

28.1 Scheme API

Festival includes a full programming language, Scheme (a variant of Lisp) as a powerful interface to its speech synthesis functions. Often this will be the easiest method of controlling Festival's functionality. Even when using other APIs they will ultimately depend on the Scheme interpreter.

Scheme commands (as s-expressions) may be simply written in files and interpreted by Festival, either by specification as arguments on the command line, in the interactive interpreter, or through standard input as a pipe. Suppose we have a file `hello.scm` containing

```
;; A short example file with Festival Scheme commands
(voice_rab_diphone) ;; select Gordon
(SayText "Hello there")
(voice_don_diphone) ;; select Donovan
(SayText "and hello from me")
```

From the command interpreter we can execute the commands in this file by loading them

```
festival> (load "hello.scm")
nil
```

Or we can execute the commands in the file directly from the shell command line

```
unix$ festival -b hello.scm
```

The `-b` option denotes batch operation meaning the file is loaded and then Festival will exit, without starting the

command interpreter. Without this option `-b` Festival will load `hello.scm` and then accept commands on standard input. This can be convenient when some initial set up is required for a session.

Note one disadvantage of the batch method is that time is required for Festival's initialisation every time it starts up. Although this will typically only be a few seconds, for saying short individual expressions that lead in time may be unacceptable. Thus simply executing the commands within an already running system is more desirable, or using the server/client mode.

Of course its not just about strings of commands, because Scheme is a fully functional language, functions, loops, variables, file access, arithmetic operations may all be carried out in your Scheme programs. Also, access to Unix is available through the `system` function. For many applications directly programming them in Scheme is both the easiest and the most efficient method.

A number of example Festival scripts are included in `examples/`. Including a program for saying the time, and for telling you the latest news (by accessing a page from the web). Also see the detailed discussion of a script example in See section [29.1 POS Example](#).

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

28.2 Shell API

The simplest use of Festival (though not the most powerful) is simply using it to directly render text files as speech. Suppose we have a file `hello.txt` containing

```
Hello world.  Isn't it excellent weather
this morning.
```

We can simply call Festival as

```
unix$ festival --tts hello.txt
```

Or for even simpler one-off phrases

```
unix$ echo "hello " | festival --tts
```

This is easy to use but you will need to wait for Festival to start up and initialise its databases before it starts to render the text as speech. This may take several seconds on some machines. A socket based server mechanism is provided in Festival which will allow a single server process to start up once and be used efficiently by multiple client programs.

Note also the use of Sable for marked up text, see section [10. XML/SGML mark-up](#). Sable allows various forms of additional information in text, such as phrasing, emphasis, pronunciation, as well as changing voices, and inclusion of external waveform files (i.e. random noises). For many application this will be the preferred interface method. Other text modes too are available through the command line by using `auto-text-mode-alist`.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

28.3 Server/client API

Festival offers a BSD socket-based interface. This allows Festival to run as a server and allow client programs to access it. Basically the server offers a new command interpreter for each client that attaches to it. The server is forked for each client but this is much faster than having to wait for a Festival process to start from scratch. Also the server can run on a bigger machine, offering much faster synthesis.

Note: the Festival server is inherently insecure and may allow arbitrary users access to your machine.

Every effort has been made to minimise the risk of unauthorised access through Festival and a number of levels of security are provided. However with any program offering socket access, like `httpd`, `sendmail` or `ftpd` there is a risk that unauthorised access is possible. I trust Festival's security enough to often run it on my own machine and departmental servers, restricting access to within our department. Please read the information below before using the Festival server so you understand the risks.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

28.3.1 Server access control

The following access control is available for Festival when running as a server. When the server starts it will usually start by loading in various commands specific for the task it is to be used for. The following variables are used to control access.

`server_port`

A number identifying the inet socket port. By default this is 1314. It may be changed as required.

`server_log_file`

If nil no logging takes place, if t logging is printed to standard out and if a file name log messages are appended to that file. All connections and attempted connections are logged with a time stamp and the name of the client. All commands sent from the client are also logged (output and data input is not logged).

`server_deny_list`

If non-nil it is used to identify which machines are not allowed access to the server. This is a list of regular expressions. If the host name of the client matches any of the regexs in this list the client is denied access. This overrides all other access methods. Remember that sometimes hosts are identified as numbers not as names.

`server_access_list`

If this is non-nil only machines whose names match at least one of the regexs in this list may connect as clients. Remember that sometimes hosts are identified as numbers not as names, so you should probably exclude the IP number of machine as well as its name to be properly secure.

`server_passwd`

If this is non-nil, the client must send this passwd to the server followed by a newline before access is given. This is required even if the machine is included in the access list. This is designed so servers for specific tasks may be set up with reasonable security.

`(set_server_safe_functions FUNCNAMELIST)`

If called this can restrict which functions the client may call. This is the most restrictive form of access, and thoroughly recommended. In this mode it would be normal to include only the specific functions the client can execute (i.e. the function to set up output, and a tts function). For example a server could call the following at set up time, thus restricting calls to only those that ``festival_client' --ttw` uses.

```
(set_server_safe_functions
  '(tts_return_to_client tts_text tts_textall Parameter.set))
```

Its is strongly recommend that you run Festival in server mode as `userid nobody` to limit the access the process will have, also running it in a `chroot` environment is more secure.

For example suppose we wish to allow access to all machines in the `CSTR` domain except for `holmes.cstr.ed.ac.uk` and `adam.cstr.ed.ac.uk`. This may be done by the following two commands

```
(set! server_deny_list '("holmes\\.cstr\\.ed\\.ac\\.uk"
                        "adam\\.cstr\\.ed\\.ac\\.uk"))
(set! server_access_list '("[^\\.]*\\.cstr\\.ed\\.ac\\.uk"))
```

This is not complete though as when DNS is not working `holmes` and `adam` will still be able to access the server (but if our DNS isn't working we probably have more serious problems). However the above is secure in that only machines in the domain `cstr.ed.ac.uk` can access the server, though there may be ways to fix machines to identify themselves as being in that domain even when they are not.

By default Festival in server mode will only accept client connections for localhost.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

28.3.2 Client control

An example client program called ``festival_client'` is included with the system that provides a wide range of access methods to the server. A number of options for the client are offered.

```
--server
    The name (or IP number) of the server host. By default this is `localhost' (i.e. the same machine you run
    the client on).
--port
    The port number the Festival server is running on. By default this is 1314.
--output FILENAME
    If a waveform is to be synchronously returned, it will be saved in FILENAME. The --ttw option uses this as
    does the use of the Festival command utt.send.wave.client. If an output waveform file is received by
    `festival_client' and no output file has been given the waveform is discarded with an error message.
--passwd PASSWD
    If a passwd is required by the server this should be stated on the client call. PASSWD is sent plus a newline
    before any other communication takes places. If this isn't specified and a passwd is required, you must enter
    that first, if the --ttw option is used, a passwd is required and none specified access will be denied.
--prolog FILE
    FILE is assumed to be contain Festival commands and its contents are sent to the server after the passwd but
    before anything else. This is convenient to use in conjunction with --ttw which otherwise does not offer any
    way to send commands as well as the text to the server.
--otype OUTPUTTYPE
    If an output waveform file is to be used this specified the output type of the file. The default is nist, but,
    ulaw, riff, ulaw and others as supported by the Edinburgh Speech Tools Library are valid. You may use
    raw too but note that Festival may return waveforms of various sampling rates depending on the sample rates
    of the databases its using. You can of course make Festival only return one particular sample rate, by using
    after_synth_hooks. Note that byte order will be native machine of the client machine if the output
    format allows it.
--ttw
    Text to wave is an attempt to make festival_client useful in many simple applications. Although you
    can connect to the server and send arbitrary Festival Scheme commands, this option automatically does what
    is probably what you want most often. When specified this options takes text from the specified file (or stdin),
    synthesizes it (in one go) and saves it in the specified output file. It basically does the following

    (Parameter.set 'Wavefiletype '<output type>)
    (tts_textall "
    <file/stdin contents>
    ")))

    Note that this is best used for small, single utterance texts as you have to wait for the whole text to be
    synthesized before it is returned.
--aucommand COMMAND
    Execute COMMAND of each waveform returned by the server. The variable FILE will be set when
    COMMAND is executed.
--async
    So that the delay between the text being sent and the first sound being available to play, this option in
    conjunction with --ttw causes the text to be synthesized utterance by utterance and be sent back in separated
    waveforms. Using --aucommand each waveform may be played locally, and when `festival_client'
    is interrupted the sound will stop. Getting the client to connect to an audio server elsewhere means the sound
    will not necessarily stop when the `festival_client' process is stopped.
--withlisp
    With each command being sent to Festival a Lisp return value is sent, also Lisp expressions may be sent from
    the server to the client through the command send_client. If this option is specified the Lisp expressions
    are printed to standard out, otherwise this information is discarded.
```

A typical example use of ``festival_client'` is

```
festival_client --async --ttw --aucommand 'na_play $FILE' fred.txt
```

This will use ``na_play'` to play each waveform generated for the utterances in ``fred.txt'`. Note the *single* quotes so that the `$` in `$FILE` isn't expanded locally.

Note the server must be running before you can talk to it. At present Festival is not set up for automatic invocations through ``inetd'` and ``/etc/services'`. If you do that yourself, note that it is a different type of interface as ``inetd'` assumes all communication goes through standard in/out.

Also note that each connection to the server starts a new session. Variables are not persistent over multiple calls to the server so if any initialization is required (e.g. loading of voices) it must be done each time the client starts or more reasonably in the server when it is started.

A PERL festival client is also available in ``festival/examples/festival_client.pl'`

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

28.3.3 Server/client protocol

The client talks to the server using s-expression (Lisp). The server will reply with and number of different chunks until either OK, is returned or ER (on error). The communication is synchronous, each client request can generate a number of waveform (WV) replies and/or Lisp replies (LP) and terminated with an OK (or ER). Lisp is used as it has its own inherent syntax that Festival can already parse.

The following pseudo-code will help defined the protocol as well as show typical use

```
fprintf(serverfd, "%s\n", s-expression);
do
    ack = read three character acknowledgemnt
    if (ack == "WV\n")
        read a waveform
    else if (ack == "LP\n")
        read an s-expression
    else if (ack == "ER\n")
        an error occurred, break;
while ack != "OK\n"
```

The server can send a waveform in an utterance to the client through the function `utt.send.wave.client`; The server can send a lisp expression to the client through the function

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

28.4 C/C++ API

As well as offerening an interface through Scheme and the shell some users may also wish to embedd Festival within their own C++ programs. A number of simply to use high level functions are available for such uses.

In order to use Festival you must include ``festival/src/include/festival.h'` which in turn will include the necessary other include files in ``festival/src/include'` and ``speech_tools/include'` you should ensure these are included in the include path for you your program. Also you will need to link your program with ``festival/src/lib/libFestival.a'`, ``speech_tools/lib/libestools.a'`, ``speech_tools/lib/libestbase.a'` and ``speech_tools/lib/libeststring.a'` as well as any

other optional libraries such as net audio.

The main external functions available for C++ users of Festival are.

```
void festival_initialize(int load_init_files,int heapsize);
    This must be called before any other festival functions may be called. It sets up the synthesizer system. The
    first argument if true, causes the system set up files to be loaded (which is normally what is necessary), the
    second argument is the initial size of the Scheme heap, this should normally be 210000 unless you envisage
    processing very large Lisp structures.
int festival_say_file(const EST_String &filename);
    Say the contents of the given file. Returns TRUE or FALSE depending on where this was successful.
int festival_say_text(const EST_String &text);
    Say the contents of the given string. Returns TRUE or FALSE depending on where this was successful.
int festival_load_file(const EST_String &filename);
    Load the contents of the given file and evaluate its contents as Lisp commands. Returns TRUE or FALSE
    depending on where this was successful.
int festival_eval_command(const EST_String &expr);
    Read the given string as a Lisp command and evaluate it. Returns TRUE or FALSE depending on where this
    was successful.
int festival_text_to_wave(const EST_String &text,EST_Wave &wave);
    Synthesize the given string into the given wave. Returns TRUE or FALSE depending on where this was
    successful.
```

Many other commands are also available but often the above will be sufficient.

Below is a simple top level program that uses the Festival functions

```
int main(int argc, char **argv)
{
    EST_Wave wave;
    int heap_size = 210000; // default scheme heap size
    int load_init_files = 1; // we want the festival init files loaded

    festival_initialize(load_init_files,heap_size);

    // Say simple file
    festival_say_file("/etc/motd");

    festival_eval_command("(voice_ked_diphone)");
    // Say some text;
    festival_say_text("hello world");

    // Convert to a waveform
    festival_text_to_wave("hello world",wave);
    wave.save("/tmp/wave.wav","riff");

    // festival_say_file puts the system in async mode so we better
    // wait for the spooler to reach the last waveform before exiting
    // This isn't necessary if only festival_say_text is being used (and
    // your own wave playing stuff)
    festival_wait_for_spooler();

    return 0;
}
```

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

28.5 C only API

A simpler C only interface example is given in `festival/examples/festival_client.c`. That interface talks to a festival server. The code does not require linking with any other EST or Festival code so is much smaller and easier to include in other programs. The code is missing some functionality but not much consider how much smaller it is.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

28.6 Java and JSAPI

Initial support for talking to a Festival server from java is included from version 1.3.0 and initial JSAPI support is included from 1.4.0. At present the JSAPI talks to a Festival server elsewhere rather than as part of the Java process itself.

A simple (Pure) Java festival client is given

```
`festival/src/modules/java/cstr/festival/Client.java' with a wraparound script in  
`festival/bin/festival_client_java'.
```

See the file `festival/src/modules/java/cstr/festival/jsapi/ReadMe` for requirements and a small example of using the JSAPI interface.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

29. Examples

This chapter contains some simple walkthrough examples of using Festival in various ways, not just as speech synthesizer

[29.1 POS Example](#) Using Festival as a part of speech tagger

[29.2 Singing Synthesis](#) Using Festival for singing

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

29.1 POS Example

This example shows how we can use part of the standard synthesis process to tokenize and tag a file of text. This section does not cover training and setting up a part of speech tag set See section [16. POS tagging](#), only how to go about using the standard POS tagger on text.

This example also shows how to use Festival as a simple scripting language, and how to modify various methods used during text to speech.

The file `examples/text2pos` contains an executable shell script which will read arbitrary ascii text from standard input and produce words and their part of speech (one per line) on standard output.

A Festival script, like any other UNIX script, it must start with the the characters `#!` followed by the name of the `festival` executable. For scripts the option `-script` is also required. Thus our first line looks like

```
#!/usr/local/bin/festival -script
```

Note that the pathname may need to be different on your system

Following this we have copious comments, to keep our lawyers happy, before we get into the real script.

The basic idea we use is that the `tts` process segments text into utterances, those utterances are then passed to a list of functions, as defined by the Scheme variable `tts_hooks`. Normally this variable contains a list of two function, `utt.synth` and `utt.play` which will synthesize and play the resulting waveform. In this case, instead, we wish to predict the part of speech value, and then print it out.

The first function we define basically replaces the normal synthesis function `utt.synth`. It runs the standard festival utterance modules used in the synthesis process, up to the point where POS is predicted. This function looks like

```
(define (find-pos utt)
  "Main function for processing TTS utterances. Predicts POS and
  prints words with their POS"
  (Token utt)
  (POS utt)
)
```

The normal text-to-speech process first tokenizes the text splitting it in to "sentences". The utterance type of these is `Token`. Then we call the `Token` utterance module, which converts the tokens to a stream of words. Then we call the `POS` module to predict part of speech tags for each word. Normally we would call other modules ultimately generating a waveform but in this case we need no further processing.

The second function we define is one that will print out the words and parts of speech

```
(define (output-pos utt)
  "Output the word/pos for each word in utt"
  (mapcar
   (lambda (pair)
     (format t "%l/%l\n" (car pair) (car (cdr pair))))
   (utt.features utt 'Word '(name pos))))
```

This uses the `utt.features` function to extract features from the items in a named stream of an utterance. In this case we want the `name` and `pos` features for each item in the `Word` stream. Then for each pair we print out the word's name, a slash and its part of speech followed by a newline.

Our next job is to redefine the functions to be called during text to speech. The variable `tts_hooks` is defined in `'lib/tts.scm'`. Here we set it to our two newly-defined functions

```
(set! tts_hooks (list find-pos output-pos))
```

So that garbage collection messages do not appear on the screen we stop the message from being outputted by the following command

```
(gc-status nil)
```

The final stage is to start the `tts` process running on standard input. Because we have redefined what functions are to be run on the utterances, it will no longer generate speech but just predict part of speech and print it to standard output.

```
(tts_file "-")
```

29.2 Singing Synthesis

As an interesting example a `singing-mode` is included. This offers an XML based mode for specifying songs, both notes and duration. This work was done as a student project by Dominic Mazzoni. A number of examples were provided in `examples/songs`. This may be run as

```
festival> (tts "doremi.xml" 'singing)
```

Each note can be given a note and a beat value

```
<?xml version="1.0"?>
<!DOCTYPE SINGING PUBLIC "-//SINGING//DTD SINGING mark up//EN"
    "Singing.v0_1.dtd"
[]>
<SINGING BPM="30">
<PITCH NOTE="G3"><DURATION BEATS="0.3">doe</DURATION></PITCH>
<PITCH NOTE="A3"><DURATION BEATS="0.3">ray</DURATION></PITCH>
<PITCH NOTE="B3"><DURATION BEATS="0.3">me</DURATION></PITCH>
<PITCH NOTE="C4"><DURATION BEATS="0.3">fah</DURATION></PITCH>
<PITCH NOTE="D4"><DURATION BEATS="0.3">sew</DURATION></PITCH>
<PITCH NOTE="E4"><DURATION BEATS="0.3">lah</DURATION></PITCH>
<PITCH NOTE="F#4"><DURATION BEATS="0.3">tee</DURATION></PITCH>
<PITCH NOTE="G4"><DURATION BEATS="0.3">doe</DURATION></PITCH>
</SINGING>
```

You can construct multi-part songs by synthesizing each part and generating waveforms, then combining them. For example

```
text2wave -mode singing americal.xml -o americal.wav
text2wave -mode singing america2.xml -o america2.wav
text2wave -mode singing america3.xml -o america3.wav
text2wave -mode singing america4.xml -o america4.wav
ch_wave -o america.wav -pc longest america?.wav
```

The voice used to sing is the current voice. Note that the number of syllables in the words must match that at run time, which means this doesn't always work cross dialect (UK voices sometimes won't work without tweaking).

This technique is basically simple, though is definitely effective. However for a more serious singing synthesizer we recommend you look at Flinger <http://cslu.cse.ogi.edu/tts/flinger/>, addresses the issues of synthesizing the human singing voice in more detail.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

30. Problems

There will be many problems with Festival, both in installation and running it. It is a young system and there is a lot to it. We believe the basic design is sound and problems will be features that are missing or incomplete rather than fundamental ones.

We are always open to suggestions on how to improve it and fix problems, we don't guarantee we'll have the time to fix problems but we are interested in hearing what problems you have.

Before you smother us with mail here is an incomplete list of general problems we have already identified

- The more documentation we write the more we realize how much more documentation is required. Most of the Festival documentation was written by someone who knows the system very well, and makes many English mistakes. A good re-write by someone else would be a good start.

- The system is far too slow. Although machines are getting faster, it still takes too long to start the system and get it to speak some given text. Even so, on reasonable machines, Festival can generate the speech several times faster than it takes to say it. But even if it is five times faster, it will take 2 seconds to generate a 10 second utterance. A 2 second wait is too long. Faster machines would improve this but a change in design is a better solution.
- The system is too big. It takes a long time to compile even on quite large machines, and its foot print is still in the 10s of megabytes as is the run-time requirement. Although we have spent some time trying to fix this (optional modules have made the possibility of building a much smaller binary) we haven't done enough yet.
- The signal quality of the voices isn't very good by today's standard of synthesizers, even given the improvement quality since the last release. This is partly our fault in not spending the time (or perhaps also not having enough expertise) on the low-level waveform synthesis parts of the system. This will improve in the future with better signal processing (under development) and better synthesis techniques (also under development).

[<] [>] [<<] [Up] [>>] [Top] [Contents] [Index] [?]

31. References

allen87

Allen J., Hunnicut S. and Klatt, D. *Text-to-speech: the MITalk system*, Cambridge University Press, 1987.

abelson85

Abelson H. and Sussman G. *Structure and Interpretation of Computer Programs*, MIT Press, 1985.

black94

Black A. and Taylor, P. "CHATR: a generic speech synthesis system.", *Proceedings of COLING-94*, Kyoto, Japan 1994.

black96

Black, A. and Hunt, A. "Generating F0 contours from ToBI labels using linear regression", *ICSLP96*, vol. 3, pp 1385-1388, Philadelphia, PA. 1996.

black97b

Black, A. and Taylor, P. "Assigning Phrase Breaks from Part-of-Speech Sequences", *Eurospeech97*, Rhodes, Greece, 1997.

black97c

Black, A. and Taylor, P. "Automatically clustering similar units for unit selection in speech synthesis", *Eurospeech97*, Rhodes, Greece, 1997.

black98

Black, A., Lenzo, K. and Pagel, V., "Issues in building general letter to sound rules.", 3rd ESCA Workshop on Speech Synthesis, Jenolan Caves, Australia, 1998.

black99

Black, A., and Lenzo, K., "Building Voices in the Festival Speech Synthesis System," unpublished document, Carnegie Mellon University, available at <http://www.cstr.ed.ac.uk/projects/festival/docs/festvox/>

breiman84

Breiman, L., Friedman, J. Olshen, R. and Stone, C. *Classification and regression trees*, Wadsworth and Brooks, Pacific Grove, CA. 1984.

campbell91

Campbell, N. and Isard, S. "Segment durations in a syllable frame", *Journal of Phonetics*, 19:1 37-47, 1991.

DeRose88

DeRose, S. "Grammatical category disambiguation by statistical optimization". *Computational Linguistics*, 14:31-39, 1988.

dusterhoff97

Dusterhoff, K. and Black, A. "Generating F0 contours for speech synthesis using the Tilt intonation theory" *Proceedings of ESCA Workshop of Intonation*, September, Athens, Greece. 1997

dutoit97

Dutoit, T. *An introduction to Text-to-Speech Synthesis* Kluwer Academic Publishers, 1997.

hunt89

Hunt, M., Zwierynski, D. and Carr, R. "Issues in high quality LPC analysis and synthesis", *Eurospeech89*, vol. 2, pp 348-351, Paris, France. 1989.

jilka96

Jilka M. *Regelbasierte Generierung natuerlich klingender Intonation des Amerikanischen Englisch*,

- Magisterarbeit, Institute of Natural Language Processing, University of Stuttgart. 1996
- moulines90*
Moulines, E, and Charpentier, N. "Pitch-synchronous waveform processing techniques for text-to-speech synthesis using diphones" *Speech Communication*, 9(5/6) pp 453-467. 1990.
- pagel98*,
Pagel, V., Lenzo, K., and Black, A. "Letter to Sound Rules for Accented Lexicon Compression", ICSLP98, Sydney, Australia, 1998.
- ritchie92*
Ritchie G, Russell G, Black A and Pulman S. *Computational Morphology: practical mechanisms for the English Lexicon*, MIT Press, Cambridge, Mass.
- vansanten96*
van Santen, J., Sproat, R., Olive, J. and Hirschberg, J. eds, "Progress in Speech Synthesis," Springer Verlag, 1996.
- silverman92*
Silverman K., Beckman M., Pitrelli, J., Ostendorf, M., Wightman, C., Price, P., Pierrehumbert, J., and Hirschberg, J "ToBI: a standard for labelling English prosody." *Proceedings of ICSLP92* vol 2. pp 867-870, 1992
- sproat97*
Sproat, R., Taylor, P, Tanenblatt, M. and Isard, A. "A Markup Language for Text-to-Speech Synthesis", *Eurospeech97*, Rhodes, Greece, 1997.
- sproat98*,
Sproat, R. eds, "Multilingual Text-to-Speech Synthesis: The Bell Labs approach", Kluwer 1998.
- sable98*,
Sproat, R., Hunt, A., Ostendorf, M., Taylor, P., Black, A., Lenzo, K., and Edgington, M. "SABLE: A standard for TTS markup." ICSLP98, Sydney, Australia, 1998.
- taylor91*
Taylor P., Nairn I., Sutherland A. and Jack M.. "A real time speech synthesis system", *Eurospeech91*, vol. 1, pp 341-344, Genoa, Italy. 1991.
- taylor96*
Taylor P. and Isard, A. "SSML: A speech synthesis markup language" to appear in *Speech Communications*.
- wwwxml97*
World Wide Web Consortium Working Draft "Extensible Markup Language (XML)Version 1.0 Part 1: Syntax", <http://www.w3.org/pub/WWW/TR/WD-xml-lang-970630.html>
- yarowsky96*
Yarowsky, D., "Homograph disambiguation in text-to-speech synthesis", in "Progress in Speech Synthesis," eds. van Santen, J., Sproat, R., Olive, J. and Hirschberg, J. pp 157-172. Springer Verlag, 1996.

[\[<\]](#) [\[>\]](#) [\[<<\]](#) [\[Up\]](#) [\[>>\]](#) [\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

32. Feature functions

This chapter contains a list of a basic feature functions available for stream items in utterances. See section [14.6 Features](#). These are the basic features, which can be combined with relative features (such as `n.` for next, and relations to follow links). Some of these features are implemented as short C++ functions (e.g. `asyl_in`) while others are simple features on an item (e.g. `pos`). Note that functional feature take precedence over simple features, so accessing and feature called "X" will always use the function called "X" even if a the simple feature call "X" exists on the item.

Unlike previous versions there are no features that are builtin on all items except `addr` (reintroduced in 1.3.1) which returns a unique string for that item (its the hex address on teh item within the machine). Features may be defined through Scheme too, these all have the prefix `lisp_`.

The feature functions are listed in the form *Relation.name* where *Relation* is the name of the stream that the function is appropriate to and *name* is its name. Note that you will not require the *Relation* part of the name if the stream item you are applying the function to is of that type.

33. Variable list

This chapter contains a list of variables currently defined within Festival available for general use. This list is automatically generated from the documentation strings of the variables as they are defined within the system, so has some chance in being up-to-date.

Cross references to sections elsewhere in the manual are given where appropriate.

34. Function list

This chapter contains a list of functions currently defined within Festival available for general use. This list is automatically generated from the documentation strings of the functions as they are defined within the system, so has some chance in being up-to-date.

Note some of the functions which have origins in the SIOD system itself are little used in Festival and may not work fully, particularly, the arrays.

Cross references to sections elsewhere in the manual are given where appropriate.

Index

Jump to: [.](#) [/](#)

[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#)

Index Entry	Section
.	
`.festivalrc'	6.3 Site initialization
/	
`/dev/audio'	23. Audio output
`/dev/audio'	23. Audio output
`/dev/dsp'	23. Audio output
A	
access strategies	21.5 Access strategies
accessing Lisp variables	27.2.1 Example 1: adding new modules
acknowledgements	3. Acknowledgements
addenda	13.2 Defining lexicons
adding new directories	27.2.3 Example 3: adding new directories

adding new LISP objects	27.2.4 Example 4: adding new LISP objects
adding new modules	27.2.1 Example 1: adding new modules
adding new voices	24.3 Defining a new voice
after analysis hooks	14.2 Utterance types
after synth hooks	14.2 Utterance types
alternate diphones	20.3 Database declaration
apostrophe s	13.8 Post-lexical rules
asynchronous synthesis	23. Audio output
atof	8.4 Scheme I/O
audio command	23. Audio output
audio command output	6.3 Site initialization
audio devices	23. Audio output
audio hardware	6.1 Requirements
audio output	23. Audio output
audio output filetype	6.3 Site initialization
audio output rate	6.3 Site initialization
audio problems	6.4 Checking an installation
audio spooler	23. Audio output
auto-text-mode-alist	9.2 Text modes
automatic selection of text mode	9.2 Text modes
automounter	6.2 Configuration

B

backtrace	8.3 Scheme Festival specifics
batch mode	7.1 Basic command line options
BEEP lexicon	13.3 Lookup process
BEEP lexicon	13.7 Available lexicons
before synth hooks	14.2 Utterance types
bug reports	7.3 Getting some help

C

C interface	28.5 C only API
C++	6.1 Requirements
CART trees	14.6 Features
CART trees	25.2 CART trees
Cascading style sheets	10. XML/SGML mark-up
catching errors	8.3 Scheme Festival specifics
catching errors in Scheme	8.3 Scheme Festival specifics
change libdir at run-time	6.3 Site initialization
client	28.3.2 Client control
client/server protocol	28.3.3 Server/client protocol
CMU lexicon	13.3 Lookup process
CMU lexicon	13.7 Available lexicons
command line options	7.1 Basic command line options
command mode	7. Quick start

command mode	7.1 Basic command line options
compiled lexicons	13.3 Lookup process
compiling a lexicon	13.2 Defining lexicons
compressing the lexicon	13.5 Building letter to sound rules
`config/config'	6.2 Configuration
configuration	6.2 Configuration
consonant clusters	21.6 Diphone selection
convert string to number	8.4 Scheme I/O
creating a lexicon	13.2 Defining lexicons
creating utterances	26.1 Labelling databases
CSLU	24.1 Current voices
CSS	10. XML/SGML mark-up
current voice	24.2.9 Resetting globals
CUVOALD lexicon	13.7 Available lexicons

D

dark l's	21.6 Diphone selection
database labelling	26.1 Labelling databases
databases	26. Building models from databases
debugging Scheme errors	8.3 Scheme Festival specifics
debugging scripts	8.3 Scheme Festival specifics
declaring text modes	9.3 Example text mode
default diphone	20.3 Database declaration
default voice	6.3 Site initialization
defSynthType	14.2 Utterance types
defUttType	14.2 Utterance types
dictionary	13. Lexicons
diphone alternates	20.3 Database declaration
diphone group files	20.4 Making groupfiles
diphone index	20.2 Generating a diphone index
diphone names	21.6 Diphone selection
diphone selection	20.6 Diphone selection
diphone synthesis	20. UniSyn synthesizer
diphone synthesis	21. Diphone synthesizer
diphone_module_hooks	21.6 Diphone selection
diphone_phone_name	21.6 Diphone selection
directory structure	27.1 The source code
display	14.7 Utterance I/O
documentation	6.1 Requirements
DSSSL	10. XML/SGML mark-up
duff intonation	18.1 Default intonation
duration	19. Duration
duration stretch	19. Duration

E

Edinburgh Speech Tools Library	3.3 Edinburgh Speech Tools Library
editline	6.1 Requirements
Emacs interface	11. Emacs interface
email mode	9.3 Example text mode
eou_tree	9.1 Utterance chunking
errors in Scheme	8.3 Scheme Festival specifics
exiting Festival	7.2 Sample command driven session
extracting features	26.2 Extracting features

F

F0 generation	18.2 Simple intonation
fclose	8.4 Scheme I/O
feature extraction	26.2 Extracting features
features	14.6 Features
features	25.2 CART trees
features	32. Feature functions
Festival relations	14.1 Utterance structure
Festival script files	7.1 Basic command line options
`festival.el'	11. Emacs interface
festival_client	28.3.2 Client control
festival_client.c	28.5 C only API
file i/o in Scheme	8.4 Scheme I/O
fixed durations	19.1 Default durations
flinger	29.2 Singing Synthesis
fopen	8.4 Scheme I/O
format	8.4 Scheme I/O
formatted output	8.4 Scheme I/O
FreeBSD	23. Audio output

G

g++	6.1 Requirements
garbage collection	29.1 POS Example
GC	29.1 POS Example
GNU g++	6.1 Requirements
GNU readline	6.1 Requirements
group files	20.4 Making groupfiles
group files	21.3 Group files
grouped diphones	20.1 UniSyn database format

H

heap size	7.1 Basic command line options
hello world	7.2 Sample command driven session
help	7.3 Getting some help
homographs	13.1 Lexical entries
homographs	15.3 Homograph disambiguation

I

i/o in Scheme	8.4 Scheme I/O
`init.scm'	6.3 Site initialization
initialization	6.3 Site initialization
Initialize	14.2 Utterance types
installation initialization	6.3 Site initialization
int accent cart tree	18.2 Simple intonation
interactive mode	7.1 Basic command line options
IntEvent labelling	26.1 Labelling databases
IntEvent labelling	26.1 Labelling databases
intonation	18. Intonation
IRIX	23. Audio output
item functions	14.5 Accessing an utterance
Items	14.1 Utterance structure

J

Java	28.6 Java and JSAPI
JSAPI	28.6 Java and JSAPI

K

Klatt duration rules	19.3 Klatt durations
--------------------------------------	--------------------------------------

L

labelling	26.1 Labelling databases
language specification	7.1 Basic command line options
laryngograph	20.1.1 Generating pitchmarks
letter to sound rules	13.2 Defining lexicons
letter to sound rules	13.4 Letter to sound rules
letter to sound rules	13.4 Letter to sound rules
letter to sound rules	24.2.2 Lexicon and LTS
lexical entries	13.1 Lexical entries
lexical stress	13.1 Lexical entries
lexicon	6.1 Requirements
lexicon	13. Lexicons
lexicon	13.7 Available lexicons
lexicon addenda	13.2 Defining lexicons
lexicon compression	13.5 Building letter to sound rules
lexicon creation	24.2.2 Lexicon and LTS
lexicon hooks	13.3 Lookup process
lexicon requirements	13.6 Lexicon requirements
linear regression	14.6 Features
linear regression	25.5 Linear regression
Linux	6.2 Configuration
Linux	23. Audio output

load-path	6.3 Site initialization
loading a waveform	14.7 Utterance I/O
loading data from files	8.4 Scheme I/O
local duration stretch	19. Duration
lookup hooks	13.3 Lookup process
LPC analysis	20.1.2 Generating LPC coefficients
LPC residual	20.1.2 Generating LPC coefficients
LTS	13.4 Letter to sound rules

M

manual	7.3 Getting some help
mapping between phones	12. Phonesets
MBROLA	22.2 MBROLA
minimal system	6.2 Configuration
MOBY lexicon	13.7 Available lexicons
modules	14.4 Utterance modules
monotone	18.1 Default intonation
mrpa lexicon	13.7 Available lexicons
multiple lexical entries	13.3 Lookup process

N

NAS	23. Audio output
netaudio	23. Audio output
new modules	27.2.1 Example 1: adding new modules
NFS	6.2 Configuration
ngrams	25.3 Ngrams
NO digits	13.6 Lexicon requirements
`nsgmls'	10.4 XML/SGML requirements

O

offline TTS	7.2 Sample command driven session
OTHER_DIRS	6.2 Configuration
output file type	6.3 Site initialization
output sample rate	6.3 Site initialization
overriding diphone names	21.6 Diphone selection
Oxford Advanced Learners' Dictionary	13.7 Available lexicons

P

Paradigm Associates	3.1 SIOD
Parameters	14.4 Utterance modules
parse-number	8.4 Scheme I/O
part of speech map	13.1 Lexical entries
part of speech tag	13.1 Lexical entries
part of speech tagging	16. POS tagging
perl	28.3.2 Client control
personal text modes	9.2 Text modes

phone maps	12. Phonesets
phoneme definitions	12. Phonesets
Phones utterance	14.3 Example utterance types
phoneset definitions	24.2.1 Phoneset
phonesets	12. Phonesets
phrase breaks	17. Phrase breaks
pitchmarking	20.1.1 Generating pitchmarks
playing an utterance	7.2 Sample command driven session
POS	13.1 Lexical entries
POS example	29.1 POS Example
POS tagging	16. POS tagging
possessives	13.6 Lexicon requirements
possessives	13.8 Post-lexical rules
post-lexical rules	13.8 Post-lexical rules
post_hooks	13.3 Lookup process
power normalisation	20.1.2 Generating LPC coefficients
pre_hooks	13.3 Lookup process
predicting stress	13.5 Building letter to sound rules
pretty printing	8.4 Scheme I/O
proclaim voice	24.3 Defining a new voice
programming	27. Programming
pronunciation	13.1 Lexical entries
PSOLA	20.5 UniSyn module selection
punctuation	15.1 Tokenizing

Q

quit	7.2 Sample command driven session
----------------------	---

R

read-eval-print loop	7.2 Sample command driven session
reading from files	8.4 Scheme I/O
readline	6.1 Requirements
redistribution	2. Copying
reducing the lexicon	13.5 Building letter to sound rules
regex	25.1 Regular expressions
regular expressions	25.1 Regular expressions
Relations	14.1 Utterance structure
Relations	27.2.2 Example 2: accessing the utterance
remote audio	23. Audio output
requirements	6.1 Requirements
resampling	14.7 Utterance I/O
rescaling a waveform	14.7 Utterance I/O
reseting globals	24.2.9 Resetting globals
residual	20.1.2 Generating LPC coefficients
restrictions	2. Copying

[resynthesis](#)
[run-time configuration](#)
[run-time configuration](#)
[rxp](#)
[14.7 Utterance I/O](#)
[6.3 Site initialization](#)
[6.3 Site initialization](#)
[10.4 XML/SGML requirements](#)

S

[Sable](#)
[SABLE DTD](#)
[Sable tags](#)
[Sable using](#)
[`Sable.v0_2.dtd'](#)
[saving relations](#)
[saving Sable waveforms](#)
[saving segments](#)
[saving the waveform](#)
[saving TTS waveforms](#)
[say-minor-mode](#)
[SayPhones](#)
[SayText](#)
[Scheme](#)
[Scheme](#)
[Scheme heap size](#)
[Scheme introduction](#)
[Scheme programming](#)
[Scheme references](#)
[script files](#)
[script programming](#)
[scripts](#)
[security](#)
[SegF0 utterance](#)
[Segment labelling](#)
[Segments utterance](#)
[selecting a phoneset](#)
[selection of diphones](#)
[selection-based synthesis](#)
[separate diphones](#)
[server mode](#)
[server security](#)
[server/client protocol](#)
[SGI](#)
[SGML](#)
[SGML parser](#)
[shell programming](#)
[silences](#)
[silences](#)
[10. XML/SGML mark-up](#)
[10.1 Sable example](#)
[10.2 Supported Sable tags](#)
[10.5 Using Sable](#)
[10.1 Sable example](#)
[14.7 Utterance I/O](#)
[10.5 Using Sable](#)
[14.7 Utterance I/O](#)
[14.7 Utterance I/O](#)
[10.5 Using Sable](#)
[11. Emacs interface](#)
[14.3 Example utterance types](#)
[7.2 Sample command driven session](#)
[3.1 SIOD](#)
[7.2 Sample command driven session](#)
[7.1 Basic command line options](#)
[8. Scheme](#)
[28.1 Scheme API](#)
[8.1 Scheme references](#)
[7.1 Basic command line options](#)
[29.1 POS Example](#)
[28.1 Scheme API](#)
[28.3.1 Server access control](#)
[14.3 Example utterance types](#)
[26.1 Labelling databases](#)
[14.3 Example utterance types](#)
[12. Phonesets](#)
[20.6 Diphone selection](#)
[22.3 Synthesizers in development](#)
[20.1 UniSyn database format](#)
[28.3 Server/client API](#)
[28.3.1 Server access control](#)
[28.3.3 Server/client protocol](#)
[23. Audio output](#)
[10. XML/SGML mark-up](#)
[10.4 XML/SGML requirements](#)
[28.2 Shell API](#)
[12. Phonesets](#)
[24.2.1 Phoneset](#)

singing	29.2 Singing Synthesis
SIOD	3.1 SIOD
`siteinit.scm'	6.3 Site initialization
smaller system	6.2 Configuration
snack	23. Audio output
source	27.1 The source code
spanish voice	24.2 Building a new voice
Spoken Text Mark-up Language	10. XML/SGML mark-up
SSML	10. XML/SGML mark-up
STML	10. XML/SGML mark-up
stress assignment	13.5 Building letter to sound rules
string to number	8.4 Scheme I/O
sun16	23. Audio output
sunaudio	23. Audio output
SunOS	6.2 Configuration
syllabification	13.3 Lookup process
Syllable labelling	26.1 Labelling databases
synthesis hooks	14.2 Utterance types
synthesis of natural utterances	14.7 Utterance I/O
synthesizing an utterance	7.2 Sample command driven session
SynthTypes	14.2 Utterance types

T

tagging	16. POS tagging
talking head	14.2 Utterance types
Target labelling	26.1 Labelling databases
TD-PSOLA	20.5 UniSyn module selection
`texi2html'	6.1 Requirements
text modes	9. TTS
text modes	9.2 Text modes
text to speech	7.2 Sample command driven session
text to wave	7.2 Sample command driven session
Text utterance	14.3 Example utterance types
text-to-speech mode	7. Quick start
text2wave	7.2 Sample command driven session
thanks	3. Acknowledgements
tokenizing	15.1 Tokenizing
tokens to words	15.2 Token to word rules
Tokens utterance	14.3 Example utterance types
tools	25. Tools
TTS	7.2 Sample command driven session
tts mode	7. Quick start
tts mode	7.1 Basic command line options
TTS processes	14.1 Utterance structure

U

ungrouped diphones	20.1 UniSyn database format
UniSyn	20. UniSyn synthesizer
unknown words	13.4 Letter to sound rules
unknown words	13.6 Lexicon requirements
unwind-protect	8.3 Scheme Festival specifics
unwind-protect	8.3 Scheme Festival specifics
us diphone init	20.3 Database declaration
user initialization	6.3 Site initialization
using Sable	10.5 Using Sable
utt.import.wave	14.7 Utterance I/O
utt.load	14.7 Utterance I/O
utt.relation functions	14.5 Accessing an utterance
utt.save	14.7 Utterance I/O
utt.save.segs	14.7 Utterance I/O
utt.save.wave	14.7 Utterance I/O
utt.synth	14.2 Utterance types
utterance	7.2 Sample command driven session
utterance	14. Utterances
utterance	14.1 Utterance structure
utterance chunking	9.1 Utterance chunking
utterance examples	14.3 Example utterance types
Utterance structure	14.1 Utterance structure
utterance types	14.2 Utterance types

V

Visual C++	6.2 Configuration
Viterbi decoder	25.4 Viterbi decoder
voice-path	24.3 Defining a new voice
voice_reset	24.2.9 Resetting globals
voices	24.1 Current voices
voxware	23. Audio output

W

wagon	25.2 CART trees
wagon	26.3 Building models
Wave utterance	14.3 Example utterance types
waveform synthesis	20. UniSyn synthesizer
whitespace	15.1 Tokenizing
wild card matching	25.1 Regular expressions
Windows 95 audio	23. Audio output
Windows NT audio	23. Audio output
Windows NT/95	6.1 Requirements
Word labelling	26.1 Labelling databases
Words utterance	14.3 Example utterance types

X

XML	10. XML/SGML mark-up
XML	10.4 XML/SGML requirements
Xwaves	14.7 Utterance I/O

Jump to: [.](#) /

[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#)

[\[Top\]](#) [\[Contents\]](#) [\[Index\]](#) [\[?\]](#)

Table of Contents

- [1. Abstract](#)
- [2. Copying](#)
- [3. Acknowledgements](#)
 - [3.1 SIOD](#)
 - [3.2 editline](#)
 - [3.3 Edinburgh Speech Tools Library](#)
 - [3.4 Others](#)
- [4. What is new](#)
- [5. Overview](#)
 - [5.1 Philosophy](#)
 - [5.2 Future](#)
- [6. Installation](#)
 - [6.1 Requirements](#)
 - [6.2 Configuration](#)
 - [6.3 Site initialization](#)
 - [6.4 Checking an installation](#)
- [7. Quick start](#)
 - [7.1 Basic command line options](#)
 - [7.2 Sample command driven session](#)
 - [7.3 Getting some help](#)
- [8. Scheme](#)
 - [8.1 Scheme references](#)
 - [8.2 Scheme fundamentals](#)
 - [8.3 Scheme Festival specifics](#)
 - [8.4 Scheme I/O](#)
- [9. TTS](#)
 - [9.1 Utterance chunking](#)
 - [9.2 Text modes](#)
 - [9.3 Example text mode](#)
- [10. XML/SGML mark-up](#)
 - [10.1 Sable example](#)
 - [10.2 Supported Sable tags](#)
 - [10.3 Adding Sable tags](#)
 - [10.4 XML/SGML requirements](#)
 - [10.5 Using Sable](#)
- [11. Emacs interface](#)

12. Phonesets

13. Lexicons

13.1 Lexical entries

13.2 Defining lexicons

13.3 Lookup process

13.4 Letter to sound rules

13.5 Building letter to sound rules

13.6 Lexicon requirements

13.7 Available lexicons

13.8 Post-lexical rules

14. Utterances

14.1 Utterance structure

14.2 Utterance types

14.3 Example utterance types

14.4 Utterance modules

14.5 Accessing an utterance

14.6 Features

14.7 Utterance I/O

15. Text analysis

15.1 Tokenizing

15.2 Token to word rules

15.3 Homograph disambiguation

15.3.1 Using disambiguators

15.3.2 Building disambiguators

16. POS tagging

17. Phrase breaks

18. Intonation

18.1 Default intonation

18.2 Simple intonation

18.3 Tree intonation

18.4 Tilt intonation

18.5 General intonation

18.6 Using ToBI

19. Duration

19.1 Default durations

19.2 Average durations

19.3 Klatt durations

19.4 CART durations

20. UniSyn synthesizer

20.1 UniSyn database format

20.1.1 Generating pitchmarks

20.1.2 Generating LPC coefficients

20.2 Generating a diphone index

20.3 Database declaration

20.4 Making groupfiles

20.5 UniSyn module selection

20.6 Diphone selection

21. Diphone synthesizer

21.1 Diphone database format

21.2 LPC databases

21.3 Group files

21.4 Diphone_Init

21.5 Access strategies

21.6 Diphone selection

22. Other synthesis methods

- [22.1 LPC diphone synthesizer](#)
- [22.2 MBROLA](#)
- [22.3 Synthesizers in development](#)
- [23. Audio output](#)
- [24. Voices](#)
 - [24.1 Current voices](#)
 - [24.2 Building a new voice](#)
 - [24.2.1 Phoneset](#)
 - [24.2.2 Lexicon and LTS](#)
 - [24.2.3 Phrasing](#)
 - [24.2.4 Intonation](#)
 - [24.2.5 Duration](#)
 - [24.2.6 Waveform synthesis](#)
 - [24.2.7 Voice selection function](#)
 - [24.2.8 Last remarks](#)
 - [24.2.9 Resetting globals](#)
 - [24.3 Defining a new voice](#)
- [25. Tools](#)
 - [25.1 Regular expressions](#)
 - [25.2 CART trees](#)
 - [25.3 Ngrams](#)
 - [25.4 Viterbi decoder](#)
 - [25.5 Linear regression](#)
- [26. Building models from databases](#)
 - [26.1 Labelling databases](#)
 - [26.2 Extracting features](#)
 - [26.3 Building models](#)
- [27. Programming](#)
 - [27.1 The source code](#)
 - [27.2 Writing a new module](#)
 - [27.2.1 Example 1: adding new modules](#)
 - [27.2.2 Example 2: accessing the utterance](#)
 - [27.2.3 Example 3: adding new directories](#)
 - [27.2.4 Example 4: adding new LISP objects](#)
- [28. API](#)
 - [28.1 Scheme API](#)
 - [28.2 Shell API](#)
 - [28.3 Server/client API](#)
 - [28.3.1 Server access control](#)
 - [28.3.2 Client control](#)
 - [28.3.3 Server/client protocol](#)
 - [28.4 C/C++ API](#)
 - [28.5 C only API](#)
 - [28.6 Java and JSAPI](#)
- [29. Examples](#)
 - [29.1 POS Example](#)
 - [29.2 Singing Synthesis](#)
- [30. Problems](#)
- [31. References](#)
- [32. Feature functions](#)
- [33. Variable list](#)
- [34. Function list](#)
- [Index](#)

Short Table of Contents

- [1. Abstract](#)
- [2. Copying](#)
- [3. Acknowledgements](#)
- [4. What is new](#)
- [5. Overview](#)
- [6. Installation](#)
- [7. Quick start](#)
- [8. Scheme](#)
- [9. TTS](#)
- [10. XML/SGML mark-up](#)
- [11. Emacs interface](#)
- [12. Phonesets](#)
- [13. Lexicons](#)
- [14. Utterances](#)
- [15. Text analysis](#)
- [16. POS tagging](#)
- [17. Phrase breaks](#)
- [18. Intonation](#)
- [19. Duration](#)
- [20. UniSyn synthesizer](#)
- [21. Diphone synthesizer](#)
- [22. Other synthesis methods](#)
- [23. Audio output](#)
- [24. Voices](#)
- [25. Tools](#)
- [26. Building models from databases](#)
- [27. Programming](#)
- [28. API](#)
- [29. Examples](#)
- [30. Problems](#)
- [31. References](#)
- [32. Feature functions](#)
- [33. Variable list](#)
- [34. Function list](#)
- [Index](#)

About this document

This document was generated by *rmannell* on *March, 15 2003* using [texi2html](#)

The buttons in the navigation panels have the following meaning:

Button	Name	Go to	From 1.2.3 go to
[<]	Back	previous section in reading order	1.2.2
[>]	Forward	next section in reading order	1.2.4

[<<]	FastBack	previous or up-and-previous section	1.1
[Up]	Up	up section	1.2
[>>]	FastForward	next or up-and-next section	1.3
[Top]	Top	cover (top) of document	
[Contents]	Contents	table of contents	
[Index]	Index	concept index	
[?]	About	this page	

where the **Example** assumes that the current position is at **Subsubsection One-Two-Three** of a document of the following structure:

- 1. Section One
 - 1.1 Subsection One-One
 - ...
 - 1.2 Subsection One-Two
 - 1.2.1 Subsubsection One-Two-One
 - 1.2.2 Subsubsection One-Two-Two
 - 1.2.3 Subsubsection One-Two-Three <== **Current Position**
 - 1.2.4 Subsubsection One-Two-Four
 - 1.3 Subsection One-Three
 - ...
 - 1.4 Subsection One-Four