The Festival Speech Synthesis System

System documentation
Edition 1.4, for Festival Version 1.4.0
17th June 1999

by Alan W Black, Paul Taylor and Richard Caley.

1 Abstract

This document provides a user manual for the Festival Speech Synthesis System, version 1.4.0.

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 develeoped 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`'

2 Copying

As we feeel the core system has reached an acceptable level of maturity from 1.4.0 the basic system is released under a free lience, 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 choosen 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

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.

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.

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.

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`'

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 synsthesis
- 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.

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 all better modularity for some of our future planned module.

- 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 impelementations. 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/’`

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

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 expected 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 there 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

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

`festival-1.4.0.tar.gz`
: Festival Speech Synthesis System source

`speech_tools-1.2.0.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_1.4.0.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 and 2.7), SunOS (4.1.3), FreeBSD 2.2, 3.x, Linux (Redhat 4.1, 5.0, 5.1, 5.2, 6.0 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

  - Sun Sparc Solaris 2.5, 2.5.1, 2.6, 2.7: GCC 2.7.2, GCC 2.8.1, SunCC 4.1, egcs 1.1.1, egcs 1.1.2

- Sun Sparc SunOS 4.1.3: GCC 2.7.2
- Intel SunOS 2.5.1: GCC 2.7.2
- FreeBSD for Intel 2.2.1, 2.2.6 and 3.x (ELF based) GCC 2.7.2.1
- Linux (2.0.30) for Intel (RedHat 4.1/5.0/5.1/5.2/6.0): GCC 2.7.2, GCC 2.7.2/egcs-1.0.2, egcs 1.1.1, egcs-1.1.2
- Windows NT 4.0: GCC 2.7.2 plus egcs (from Cygnus GNU win32 b19), Visual C++ PRO v5.0

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

We still recommend GCC 2.7.2 which we use as our standard compiler. It is (mostly) standard across platforms and compiles faster and produces better code than any of the other compilers we've used.

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 Chapter 23 [Audio output], page 103.

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`'.

Most of the related software not part of the Festival distribution has been made available in

`'ftp://ftp.cstr.ed.ac.uk/pub/festival/extras/'`

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/
```

6.2 Configuration

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

Before compilation of Festival it is necessary to configure your implementation to be aware of the environment it is being compiled in. Specifically it must know the names of various local programs, such as your compiler; directories were local libraries are held, and choices for various options about sub-systems it is to use. In most cases this can be done automatically if your system is a supported, otherwise it may be necessary to edit a few lines.

All compilation information is set in a local per installation file called '`config/config`'. You should copy the example one, mark it writable and edit it according to your local set up.

```
cd config/
cp config-dist config
chmod +w config
```

'`config/config`' is included by all '`Makefiles`' in the system and therefore should be the only place machine specific information need be changed. Note that all '`Makefiles`' define the variable `TOP` to allow appropriate relative addressing of directories within the '`Makefiles`' and their included files.

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.

To check your configuration type in the '`festival/`' directory.

```
gnumake info
```

If that seems fine, compile the system with

```
gnumake
```

On completion you can check the system with

```
gnumake test
```

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 causes warnings like "implicit definition of fprintf". These are harmless.

Under Sun's CC compiler a number of warnings are given about not being able to find source, particularly for operator `<<` and some `==` operators. It is unclear why this should be a warning as the code exists in other files deliberately for modularity purposes and should not be visible in these files anyway. These warnings 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 Professonal 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 cygnus 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 instalation 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'.

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 your 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 Festivals 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.

For example a particular installation of Festival may set Spanish as the default language by adding

```
(language_spanish)
```

in 'siteinit.scm', while a user may wish their version to use Welsh by default. In this case they would add

```
(language_welsh)
```

to their '.festivalrc' in their home directory.

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 1.4.0:release June 1999
Copyright (C) University of Edinburgh, 1996-1999. 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 1.4.0:release June 1999
Copyright (C) University of Edinburgh, 1996-1999. 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.

6.5 Y2K

Festival comes with *no* warranty therefore we will not make any legal statement about the performance of the system. However a number of people have ask about Festival and Y2K compliance, and we have decided to make some comments on this.

Every effort has been made to ensure that Festival will continue running as before into the next millenium. However even if Festival itself has no problems it is dependent on the operating system environment it is running in. During compilation dates on files are important and the compilation process may not work if your machine cannot assign (reasonable) dates to new files. At run time there is less dependence on system dates and times. Specifically times are used in generation of random numbers (where only relative time is important) and as time stamps in log files when festival runs in server mode, thus we feel it is unlikely there will be any problems.

However, as a speech synthesizer, Festival must make explicit decisions about the pronunciation of dates in the next two decades when people themselves have not yet made such decisions. Most people are still unsure how to read years written as '01, '04, '12, 00s, 10s, (cf. '86, 90s). It is interesting to note that while there is a convenient short name for the last decade of the twentieth century, the `"ninties"` there is no equivalent name for the first decade of the twenty-first century (or the second). In the mean time we have made reasonable decisions about such pronunciations.

Once people have themselves become Y2K compliant and decided what to actually call these years, if their choices are different from how Festival pronounces them we reserve the right to change how Festival speaks these dates to match their belated decisions. However as we do not give out warranties about compliance we will not be requiring our users to return signed Y2K compliant warranties about their own compliance either.

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

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], page 138. Note server mode may be unsafe and allow unauthorised access to your machine, be sure to read the security recommendations in Section 28.3 [Server/client API], page 138

`--script scriptfile`
: Run scriptfile as a Festival script file. This is similar to 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.

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 1.4.0:release June 1999
Copyright (C) University of Edinburgh, 1996-1999. 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], page 65), 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.

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 Chapter 23 [Audio output], page 103.

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.

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

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/'

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 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.

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

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.

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 formating 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 reseting 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 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 with 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 you're 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.

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.feat 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.feat 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 Labls, 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 negligable 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

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 explicit specified as can new pronunciations of words. The last sentence specifies some external filenames to play at that point.

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 announces 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 particualr installation of Festival "Some text in .." is said instead and the section is ommitted.

`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 particalur 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 tags 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 this tag is an emtpy tag and must include the closing part within itsefl 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 explcitily 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 indeitnfication of the enclose tokens/text. The attribute `MODE` cand take any of the following a 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 speficied 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 enclose 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 utetrance 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.

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 lamnguages 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 Chapter 11 [Emacs interface], page 45

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 audo 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`).

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.

12 Phonesets

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

A phoneset 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 phoneset. 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 phoneset 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 accessedwith 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))
```

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.

13.1 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 markigns 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], page 51 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], page 52, 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 @ l) 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.

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], page 52.

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 ei1 b l))
("suspicious" nil (s @ s p i1 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.

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 or 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.

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 latter 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/_epsilons_.
- 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 auxiluary 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 ei 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 constract 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> (cummulate-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 `cummulate-pairs` again. Incrementally add to the allowable pairs until the number of failures becomes accceptable. 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 pronunced 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 `_epsilons_`. 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 indentify 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 signficantly 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 constructrure 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 significnat amount of time to run, about 6 hours on a Sun Ultra 140.

Once the models are created the 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 tress.

```
(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 diferences, 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 stree 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. If 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 stree assignment system) can correct 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 signifcant, 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 diccussion of using the above technique as a dictionary compression method is discussed in *pagel98*. 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 dicussed more fully in *black98*.

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.

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'. 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.

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 possesive 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.feat
                             seg "R:SylStructure.parent.parent.name"))
         (if (string-equal "a" (item.feat seg 'ph_vlng))
             (if (and (member_string (item.feat seg 'p.ph_ctype)
                                     '(f a))
                      (not (member_string
                            (item.feat seg "p.ph_cplace")
                            '(d b g))))
                 t;; don't delete schwa
                 (item.delete seg))
             (if (string-equal "-" (item.feat seg "p.ph_cvox"))
                 (item.set_name seg "s")))));; from "z"
   (utt.relation.items utt 'Segment))
  utt)
```

14 Utterances

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

14.1 Utterance structure

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

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

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

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

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

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

Prosodic phrasing
: Chunk utterance into prosodic phrases.

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

Intonational accents
: Assign intonation accents to approrpiate syllables.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

14.2 Utterance types

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

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

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

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

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

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

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

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

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

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

  (print "select")
  (simple_diphone_select utt)

  (print "join")
  (cut_unit_join utt)

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

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

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

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

14.3 Example utterance types

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

`Text` Raw text as a string.

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

`Words` A list of words

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

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

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

Note: the use of the tone feature requires an intonation mode that supports it.

Any feature and value named in the input will be added to the Word item.

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

```
(Utterance
 Phrase
```

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

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

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

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

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

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

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

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

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

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

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

14.4 Utterance modules

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

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

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

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

14.5 Accessing an utterance

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
(define (syntax_relation_punc p)
  (if (string-equal "punc" (item.feat (item.daughter2 p) "pos"))
```

```
            (item.move_tree (item.daughter1 p) p)
        (mapcar syntax_remove_punc (item.daughters p))))
```

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

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

With the above functions quite elaborate utterance manipulations can be performed. For example in post-lexical rules where modifications to the segments are required based on the words and their context. See Section 13.8 [Post-lexical rules], page 60 for an example of using various utterance access functions.

14.6 Features

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

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

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

`n.`
: next

`p.`
: previous

`nn.`
: next next

`pp.`
: previous

`parent.`

`daughter1.`
: first daughter

`daughter2.`
: second daughter

`daughtern.`
: last daughter

`first.` most previous item

`last.` most next item

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

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

'`stress`' This item's lexical stress

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

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

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

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

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

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

A list of all feature functions is given in an appendix of this document. See Chapter 32 [Feature functions], page 151. 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], page 126 for an example of extracting sets of features from a database for use in building stochastic models.

14.7 Utterance I/O

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

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

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

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

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

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

15 Text analysis

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.

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
```

Becuase 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.feat token "n.name") ".*illion.?"))
   (builtin_english_token_to_words token (string-after name "$")))
  ((and (string-matches (item.feat token "p.name")
                        "\\$[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.

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 Chapter 16 [POS tagging], page 79. But even part of speech isn't sufficient in a number of cases. Words such as "bass", "wind", "bow" etc cannot by 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.

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.feat token "token_pos") "street")
     (if (string-matches name "[dD][rR]")
         (list "drive")
         (list "street"))
     (if (string-matches name "[dD][rR]")
         (list "doctor")
         (list "saint"))))
```

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 the 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 a list of pairs consisting of a list of tags to be mapped and the new tag they are to be mapped to.

Note is it 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.

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

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)
```

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))))
```

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.

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 as described below See Section 25.5 [Linear regression], page 123. Three models are used in the variables `f0_lr_start`, `f0_lr_mid` and `f0_lr_end`.

18.4 Tilt intonation

Tilt description to be inserted.

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.feat syl 'syllable_start))
        (end (item.feat syl 'syllable_end)))
    (if (equal? (item.feat 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)))
```

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 folwoing variables.

```
(Parameter.set 'Default_Topline 110)
(Parameter.set 'Default_Start_Baseline 87)
(Parameter.set 'Default_End_Baseline 83)
(Parameter.set 'Current_Topline (Parameter.get 'Default_Topline))
(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 teh command line on 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 the synthesized form of these utterance in Xwaves. Start an Xwaves and an Xlabeller and call the function `display` on the synthesized utterance.

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` wchi will return the combined gloabel 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

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).

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.

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`'.

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.

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.

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 databse 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.

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-ll   &aa-l
```

This states that the diphone `aa-ll` should actually use the diphone `aa-l`. Note they are a number of ways to specify alternates for missing diphones an 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.

20.3 Database declaration

There two major types of database *grouped* and *ungrouped*. Grouped databases come as a single file containing the diphone index, coeficinets and residuals for the diphones. This is the standard way databases are distributed as voices in Festoval. Ungrouped access diphones from individual files and is designed as a method for debugging and testing databases before distribution. Using ungrouped dataabse 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 laded database.

`index_file`
: A filename name containing either a diphone index, as descripbed 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 extention for coefficient files, typically `".lpc"` for LPC file and `".pm"` for pitchmark files.

`sig_ext` The extention 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 its 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 is 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 is 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)
```

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 diphone sin the dtabaase 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 choosen 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 that 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.

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.

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 is 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.

21 Diphone synthesizer

NOTE: use of this diphone synthesis is depricated 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

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. Section 21.2 [LPC databases], page 96

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).

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.

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 runtime 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.

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 us the silence to silence diphone. No mater 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’`

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.

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 `ll`. 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 (`ll`) 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 ll."
  (mapcar
   (lambda (seg)
     (if (and (string-equal "l" (item.name seg))
              (string-equal "+" (item.feat seg "p.ph_vc"))
              (item.relation.prev seg "SylStructure"))
      (item.set_feat seg "diphone_phone_name" "ll")))
   (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, `ll` to `l` as sometimes the dark l diphone doesn't actually exist in the database.

22 Other synthesis methods

Festival supports a number of other synthesis systems

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

```
lib/voices/english/don_diphone/festvox/don_diphone.scm
```

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 seleciton synthesis module is included in '`modules/clunits/`' it is an implementation of *black97c*. It is included for people wishing to continue reserach 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 Chapter 6 [Installation], page 15

'`/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 treatement 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")
```

If Netaudio is not available and you need to play audio on a machine different from teh 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 importnatly 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 be 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.

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 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 Amercian 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 Amercian 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 Amercian 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 anlysis for Klingon still requires some work (If anyone has access to a good Klingon continous speech corpora please let us know.)

Pointers and examples of voices developed at CSTR and elsewhere will be posted on the Festival home page.

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.

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
   (vlng 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  + l 3 1 - 0 0 -)
   (e  + l 2 1 - 0 0 -)
   (i  + l 1 1 - 0 0 -)
   (o  + l 3 3 - 0 0 -)
   (u  + l 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.

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.

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))))))
```

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)))
```

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)
))
```

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 "#-#")))
```

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)
```

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], page 117 for a description of methods for adding new voices. For testing purposes we can explciitly 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.

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 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 for 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 probabaly 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.

24.3 Defining a new voice

As there are a number of voices available for Festival and they may or may not exists 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` conatins 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 Amercan 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 langauge directoriy 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 symbole link in '`festival/lib/voices/english/`' ro '`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 convetions 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 definiton and a comment. For example a definition of voice can be done explicitly by

```
(autooad 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 intellegent 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 taks 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 proclaimation 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 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 of problem of decide 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 actaully 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 stat 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 in to 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

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 regualr 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 preceeding 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 specificed 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 actuall only one backslash is needed before a character to escape it but becuase these expressions are most often contained with Scheme or C++ strings, the escpae mechanaism for those strings requires that backslash itself be escaped, hence you will most often be required to type two backslashes.

Some example may help in enderstanding 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 preceeding 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]+`
: mathes any non-empty string which doesn't conatin 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 macthes the string and `nil` otherwise.

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], page 71).

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.

25.3 Ngrams

Bigram, trigrams, and general ngrams are used in the part of speech tagger and the phrase break predicter. 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'.

25.4 Viterbi decoder

Another common tool is a Viterbi decoder. This C++ Class is defined in the speech tools library 'speech_tooks/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 decoeder 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 probilities).

`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".

`wfstmname`
: 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 candidiate 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 probabilties 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 cands with rprobs
 (cadr
  (lex.lookup (item.name w) nil)))
```

The returned candidate list would look somthing 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.

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], page 71) 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.

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

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 ifnromation 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 hte 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.

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 recommedn 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. Extar 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 trated 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.

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.

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

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 simpel 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/arc/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 multilingual, 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 the the Stochastic Context Free Grammar parser in the speech tools library.

'./src/modules/diphone'
: An optional module contain the previouslty used diphone synthsizer.

'./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 inteneded 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.

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++.

27.2.1 Example 1

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 knew 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 `phone_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.

27.2.2 Example 2

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;
    }
```

27.2.3 Example 3

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 with 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.

28 API

If you wish to use Festival within some other application there are a number of possible interfaces.

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 API's 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], page 145.

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 Chapter 10 [XML/SGML mark-up], page 39. 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`.

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.

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`.

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 my 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`'

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 normallly 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;
}
```

28.5 C only API

A simpler C only interface example is given inf '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.

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.

29 Examples

This chapter contains some simple walkthrough examples of using Festival in various ways, not just as speech synthesizer

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 Chapter 16 [POS tagging], page 79), 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 "-")
```

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 some one 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 time 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).

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 Acedemic 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.

32 Feature functions

This chapter contains a list of a basic feature functions available for stream items in utterances. See Section 14.6 [Features], page 71. 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 precidence 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.

`ANY.addr` Returned by popular demand, returns the address of given item that is guaranteed unique for this session.

`ANY.lisp_*`
: Apply Lisp function named after lisp_. The function is called with an stream item. It must return an atomic value. This method may be inefficient and is primarily desgined to allow quick prototyping of new feature functions.

`Intonation.lisp_last_tilt_accent`
: Returns the most recent tilt accent.

`Intonation.lisp_last_tilt_boundary`
: Returns the most recent tilt boundary.

`Intonation.lisp_next_tilt_accent`
: Returns the next tilt accent.

`Intonation.lisp_next_tilt_boundary`
: Returns the next tilt boundary.

`Intonation.peak_anchor_segment_type ie`
: Determines whether the segment anchor for a peak is the first consonant of a syl - C0 -, the vowel of a syl - V0 -, or segments after that - C1->X,V1->X. If the segment is in a following syl, the return value will be preceded by a 1 - e.g. 1V1

`Segment.diphone_phone_name`
: This is produced by the diphone module to contain the desired phone name for the desired diphone. This adds things like _ if part of a consonant or $ to denote syllable boundaries. These are generated on a per voice basis by function(s) specified by diphone_module_hooks. Identification of dark ll's etc. may also be included. Note this is not necessarily the name of the diphone selected as if it

is not found some of these characters will be removed and fall back values will be used.

`Segment.lisp_pos_in_syl seg`
: Finds the position in a syllable of a segment - returns a number.

`Segment.ph_*`
: Access phoneset features for a segment. This definition covers multiple feature functions where ph_ may be extended with any features that are defined in the phoneset (e.g. vc, vlng, cplace etc.).

`Segment.pos_in_syl`
: The position of this segment in the syllable it is related to. The index counts from 0. If this segment is not related to a syllable this returns 0.

`Segment.seg_coda_fric`
: Returns 1 if coda of the syllable this segment is in contains a fricative. 0 otherwise.

`Segment.seg_onset_stop`
: Returns 1 if onset of the syllable this segment is in contains a stop. 0 otherwise.

`Segment.seg_onsetcoda`
: Returns onset if this segment is before the vowel in the syllable it is contained within. Returns coda if it is the vowel or after. If the segment is not in a syllable it returns onset.

`Segment.seg_pitch`
: Pitch at the middle of this segment.

`Segment.segment_duration`
: The duration of the given stream item calculated as the end of this item minus the end of the previous item in the Segment relation.

`Segment.segment_end`
: The end time of the given segment.

`Segement.segment_mid`
: The middle time of the given segment.

`Segement.segment_start`
: The start time of the given segment.

`Segment.syl_final`
: Returns 1 if this segment is the last segment in the syllable it is related to, or if it is not related to any syllable.

`Segment.syl_initial`
: Returns 1 if this segment is the first segment in the syllable it is related to, or if it is not related to any syllable.

`Syllable.accented`
: Returns 1 if syllable is accented, 0 otherwise. A syllable is accented if there is at least one IntEvent related to it.

`Syllable.asyl_in`
: Returns number of accented syllables since last phrase break, not including this one. Accentedness is as defined by the syl_accented feature.

`Syllable.asyl_out`
: Returns number of accented syllables to the next phrase break, not including this one. Accentedness is as defined by the syl_accented feature.

`Syllable.last_accent`
: Returns the number of syllables since last accented syllable.

`Syllable.lisp_last_stress`
: Number of syllables from previous stressed syllable. 0 if this syllable is stressed. It is effectively assumed that the syllable before the first syllable is stressed.

`Syllable.lisp_next_stress`
: Number of syllables to next stressed syllable. 0 if this syllable is stressed. It is effectively assumed the syllable after the last syllable is stressed.

`Syllable.lisp_tilt_accent`
: Returns "a" if there is a tilt accent related to this syllable, 0 otherwise.

`Syllable.lisp_tilt_accented`
: Returns 1 if there is a tilt accent related to this syllable, 0 otherwise.

`Syllable.lisp_tilt_boundaried`
: Returns 1 if there is a tilt boundary related to this syllable, 0 otherwise.

`Syllable.lisp_tilt_boundary`
: Returns boundary label if there is a tilt boundary related to this syllable, 0 otherwise.

`Syllable.lisp_time_to_next_vowel syl`
: The time from vowel_start to next vowel_start

`Syllable.next_accent`
: Returns the number of syllables to the next accented syllable.

`Syllable.old_syl_break`
: Like syl_break but 2 and 3 are promoted to 4 (to be compatible with some older models.

`Syllable.pos_in_word`
: The position of this syllable in the word it is related to. The index counts from 0. If this syllable is not related to a word then 0 is returned.

`Syllable.position_type`
: The type of syllable with respect to the word it it related to. This may be any of: single for single syllable words, initial for word initial syllables in a poly-syllabic word, final for word final syllables in poly-syllabic words, and mid for syllables within poly-syllabic words.

`Syllable.ssyl_in`
: Returns number of stressed syllables since last phrase break, not including this one.

`Syllable.ssyl_out`
: Returns number of stressed syllables to next phrase break, not including this one.

`Syllable.stress`
: The lexical stress of the syllable as specified from the lexicon entry corresponding to the word related to this syllable.

`Syllable.sub_phrases`
: Returns the number of non-major phrase breaks since last major phrase break. Major phrase breaks are 4, as returned by syl_break, minor phrase breaks are 2 and 3.

`Syllable.syl_accent`
: Returns the name of the accent related to the syllable. NONE is returned if there are no accents, and multi is returned if there is more than one.

`Syllable.syl_break`
: The break level after this syllable. Word internal is syllables return 0, non phrase final words return 1. Final syllables in phrase final words return the name of the phrase they are related to. Note the occasional `"-"` that may appear of phrase names is removed so that this feature function returns a number in the range 0,1,2,3,4.

`Syllable.syl_coda_type`
: Return the van Santen and Hirschberg classification. -V for unvoiced, +V-S for voiced but no sonorants, and +S for sonorants.

`Syllable.syl_codasize`
: Returns the number of segments after the vowel in this syllable. If there is no vowel in the syllable this will return the total number of segments in the syllable.

`Syllable.syl_endpitch`
: Pitch at the end of this syllable.

`Syllable.syl_in`
: Returns number of syllables since last phrase break. This is 0 if this syllable is phrase initial.

`Syllable.syl_midpitch`
: Pitch at the mid vowel of this syllable.

`Syllable.syl_numphones`
: Returns number of phones in syllable.

`Syllable.syl_onset_type`
: Return the van Santen and Hirschberg classification. -V for unvoiced, +V-S for voiced but no sonorants, and +S for sonorants.

`Syllable.syl_onsetsize`
: Returns the number of segments before the vowel in this syllable. If there is no vowel in the syllable this will return the total number of segments in the syllable.

`Syllable.syl_out`
: Returns number of syllables to next phrase break. This is 0 if this syllable is phrase final.

`Syllable.syl_pc_unvox`
: Percentage of total duration of unvoiced segments from start of syllable. (i.e. percentage to start of first voiced segment)

`Syllable.syl_startpitch`
: Pitch at the start of this syllable.

`Syllable.syl_vowel`
: Returns the name of the vowel within this syllable. Note this is not the general form you probably want. You can't refer to ph_* features of this. Returns "novowel" is no vowel can be found.

`Syllable.syl_vowel_start`
: Start position of vowel in syllable. If there is no vowel the start position of the syllable is returned.

`Syllable.syllable_duration`
: The duration of the given stream item calculated as the end of last daughter minus the end of previous item in the Segment relation of the first duaghter.

`Syllable.word_end`
: The end time of the given syllable.

`Syllable.syllable_start`
: The start time of the given syllable.

`Syllable.tobi_accent`
: Returns the ToBI accent related to syllable. ToBI accents are those which contain a *. NONE is returned if there are none. If there is more than one ToBI accent related to this syllable the first one is returned.

`Syllable.tobi_endtone`
: Returns the ToBI endtone related to syllable. ToBI end tones are those IntEvent labels which contain a % or a - (i.e. end tones or phrase accents). NONE is returned if there are none. If there is more than one ToBI end tone related to this syllable the first one is returned.

`Syllable.lisp_get_onset_length`
: Length from start of syllable to start of vowel.

`Syllable.lisp_get_rhyme_length`
: Length from start of the vowel to end of syllable.

`SylStructure.lisp_length_to_last_seg`
: Length from start of the vowel to start of last segment of syllable.

`SylStructure.lisp_num_postvocalic_c`
: Finds the number of postvocalic consonants in a syllable.

`SylStructure.sonority_scale_coda syl`
: Returns value on sonority scale (1 -6, where 6 is most sonorous) for the coda of a syllable, based on least sonorant portion.

`SylStructure.sonority_scale_onset syl`
: Returns value on sonority scale (1 -6, where 6 is most sonorous) for the onset of a syllable, based on least sonorant portion.

`SylStructure.lisp_syl_numphones syl`
: Finds the number segments in a syllable.

`SylStructure.vowel_frontness syl`
: Classifies vowels as front, back or mid

`SylStructure.lisp_vowel_height syl`
: Classifies vowels as high, low or mid

`SylStructure.vowel_length syl`
: Returns the df.length feature of a syllable's vowel

`Token.prepunctuation`
: Preceeding puctuation symbol found before token in original string/file.

`Token.punc`
: Succeeding punctuation symbol found after token in original string/file.

`Token.whitespace`
: Whitespace found before token in original string/file.

`Word.blevel`
: A crude translation of phrase break into ToBI like phrase level. Values may be 0,1,2,3,4.

`Word.cap`
: Returns 1 if this word starts with a capital letter, 0 otherwise.

`Word.content_words_in`
: Number of content words from start this phrase.

`Word.content_words_out`
: Number of content words to end of this phrase.

`Word.contentp`
: Returns 1 if this word is a content word as defined by gpos, 0 otherwise.

`Word.gpos`
: Returns a guess at the part of speech of this word. The lisp a-list guess_pos is used to load up this word. If no part of speech is found in there "content" is returned. This allows a quick efficient method for part of speech tagging into closed class and content words.

`Word.n_content`
: Next content word. Note this doesn't use the standard n. notation as it may have to search a number of words forward before finding a non-function word. Uses gpos to define content/function word distinction. This also works for Tokens.

`Word.nn_content`
: Next next content word. Note this doesn't use the standard n.n. notation as it may have to search a number of words forward before finding the second non-function word. Uses gpos to define content/function word distinction. This also works for Tokens.

`Word.num_break`
: 1 if this is the last word in a numeric token and it is followed by a numeric token.

`Word.p_content`
: Previous content word. Note this doesn't use the standard p. notation as it may have to search a number of words backward before finding the first non-function word. Uses gpos to define content/function word distinction. This also works for Tokens.

`Word.pbreak`
: Result from statistical phrasing module, may be B or NB denoting phrase break or non-phrase break after the word.

`Word.pbreak_score`
: Log likelihood score from statistical phrasing module, for pbreak value.

`Word.pos`
: Part of speech tag value returned by the POS tagger module.

`Word.pos_in_phrase`
: The position of this word in the phrase this word is in.

`Word.pos_score`
: Part of speech tag log likelihood from Viterbi search.

`Word.pp_content`
: Previous previous content word. Note this doesn't use the standard p.p. notation as it may have to search a number of words backward before finding the first non-function word. Uses gpos to define content/function word distinction. This also works for Tokens.

`Word.word_break`
: The break level after this word. Non-phrase final words return 1 Phrase final words return the name of the phrase they are in.

`Word.word_duration`
: The duration of the given stream item. This is defined as the end of last segment in the last syllable (via the SylStructure relation) minus the segment immediate preceeding the first segment in the first syllable.

`Word.word_end`
: The end time of the given word.

`Word.word_numsyls`
: Returns number of syllables in a word.

`Word.word_start`
: The start time of the given word.

`Word.words_out`
Number of words to end of this phrase.

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 int he manual are given where appropriate.

`!`
: In interactive mode, this variable's value is the return value of the previously evaluated expression.

`*module-descriptions*`
: An association list recording the description objects for proclaimed modules.

`*ostype*`
: Contains the name of the operating system type that Festival is running on, e.g. SunOS5, FreeBSD, linux etc. The value is taken from the Makefile variable OSTYPE at compile time.

`*properties*`
: Array for holding symbol property lists.

`after_analysis_hooks`
: List of functions to be applied after analysis and before synthesis.

`after_synth_hooks`
: List of functions to be applied after all synthesis modules have been applied. This is primarily designed to allow waveform manipulation, particularly resampling and volume changes.

`auto-text-mode-alist`
: Following Emacs' auto-mode-alist thios provides a mechanism for auto selecting a TTS text mode based on the filename being analyzed. Its format is exactly the same as Emacs in that it consists of an alist of dotted pairs of regular expression and text mode name.

`before_synth_hooks`
: List of functions to be run on synthesised utterances before synthesis starts.

`default-voice-priority-list`
: List of voice names. The first of them available becomes the default voice.

`default_access_strategy`
: How to access units from databases.

`default_after_analysis_hooks`
: The default list of functions to be run on all synthesized utterances after analysis but before synthesis.

`default_after_synth_hooks`
: The default list of functions to be run on all synthesized utterances after Wave_Synth. This will normally be nil but if for some reason you need to change the gain or rescale *all* waveforms you could set the function here, in your siteinit.scm.

`default_before_synth_hooks`
: The default list of functions to be run on all synthesized utterances before synthesis starts.

`diphone_module_hooks`
: A function or list of functions that will be applied to the utterance at the start of the diphone module. It can be used to map segment names to those that will be used by the diphone database itself. Typical use specifies _ and $ for consonant clusters and syllable boundaries, mapping to dark ll's etc. Reduction and tap type phenomena should probabaly be done by post lexical rules though the distinction is not a clear one.

`duffint_params`
: Default parameters for Default (duff) intonation target generation. This is an assoc list of parameters. Two parameters are supported start specifies the start F0 in Hertz for an utterance, and end specifies the end.

`editline_histsize`
: The number of lines to be saved in the users history file when a Festival session ends. The histfile is ".festival_history" in the users home directory. Note this value is only checked when the command interpreter is started, hence this should be set in a user's ".festivalrc" or system init file. Reseting it at the command interpreter will have no effect.

`editline_no_echo`
: When running under Emacs as an inferior process, we don't want to echo the content of the line, only the prompt.

`english_homographs`
: A list of tokens that are dealt with by a homograph disambiguation tree in english_token_pos_cart_trees.

`english_phr_break_params`
: Parameters for English phrase break statistical model.

`eou_tree`
: End of utterance tree. A decision tree used to determine if the given token marks the end of an utterance. It may look one token ahead to do this. [see Section 9.1 [Utterance chunking], page 33]

`etc-path`
: A list of directories where binaries specific to Festival may be located. This variable is automatically set to LIBDIR/etc/OSTYPE/ and that path is added to the end of the UNIX PATH environment variable.

`festival_version`
: A string containing the current version number of the system.

`festival_version_number`
: A list of major, minor and subminor version numbers of the current system. e.g. (1 0 12).

`FP_duration`
: In using Fixed_Prosody as used in Phones type utterances and hence SayPhones, this is the fix value in ms for phone durations.

`FP_F0` In using Fixed_Prosody as used in Phones type utterances and hence SayPhones, this is the value in Hertz for the monotone F0.

`guess_pos`
: An assoc-list of simple part of speech tag to list of words in that class. This basically only contains closed class words all other words may be assumed to be content words. This was built from information in the f2b database and is used by the ffeature gpos.

`home-directory`
: Place looked at for .festivalrc etc.

`hush_startup`
: If set to non-nil, the copyright banner is not displayed at start up.

`int_tilt_params`
: Parameters for tilt intonation model.

`lexdir` The directory where the lexicon(s) are, by default.

`libdir` The pathname of the run-time libary directory. Note reseting is almost definitely not what you want to do. This value is automatically set at start up from the value specifed at compile-time or the value specifed with –libdir on the command line. A number of other variables depend on this value.

`load-path`
: A list of directories containing .scm files. Used for various functions such as load_library and require. Follows the same use as EMACS. By default it is set up to the compile-time library directory but may be changed by the user at run time, by adding a user's own library directory or even replacing all of the standard library. [see Section 6.3 [Site initialization], page 19]

`manual-browser`
: The Unix program name of your Netscape Navigator browser. [see Section 7.3 [Getting some help], page 28]

`manual-url`
: The default URL for the Festival Manual in html format. You may reset this to a file://.../... type URL on you're local machine. [see Section 7.3 [Getting some help], page 28]

`mbrola_database`
: The name of the MBROLA database to usde during MBROLA Synthesis.

`mbrola_progname`
: The program name for mbrola.

`Param` A feature set for arbitrary parameters for modules.

`pbreak_ngram_dir`
: The directory containing the ngram models for predicting phrase breaks. By default this is the standard library directory.

`phr_break_params`
: Parameters for phrase break statistical model. This is typcal set by a voice selection function to the parameters for a particular model.

`pos_map`
: A reverse assoc list of predicted pos tags to some other tag set. Note using this changes the pos tag loosing the actual predicted value. Rather than map here you may find it more appropriate to map tags sets locally in the module sthat use them (e.g. phrasing and lexicons).

`pos_model_dir`
: The directory contains the various models for the POS module. By default this is the same directory as lexdir. The directory should contain two models: a part of speech lexicon with reverse log probabilities and an ngram model for the same part of speech tag set.

`pos_ngram_name`
: The name of a loaded ngram containing the a posteriori ngram model for predicting part of speech. The a priori model is held as a lexicon call poslex.

`pos_p_start_tag`
: This variable's value is the tag most likely to appear before the start of a sentence. It is used when looking for pos context before an utterance. Typically it should be some type of punctuation tag.

`pos_pp_start_tag`
: This variable's value is the tag most likely to appear before pos_p_start_tag and any position preceding that. It is typically some type of noun tag. This is used to provide pos context for early words in an utterance.

`pos_supported`
: If set to non-nil use part of speech prediction, if nil just get pos information from the lexicon.

`postlex_mrpa_r_cart_tree`
: For remove final R when not between vowels.

`postlex_rules_hooks`
: A function or list of functions which encode post lexical rules. This will be voice specific, though some rules will be shared across languages.

`postlex_vowel_reduce_cart_tree`
: CART tree for vowel reduction.

`postlex_vowel_reduce_cart_tree_hand`
: A CART tree for vowel reduction. This is hand-written.

`postlex_vowel_reduce_table`
: Mapping of vowels to their reduced form. This in an assoc list of phoneset name to an assoc list of full vowel to reduced form.

`provided`
: List of file names (omitting .scm) that have been provided. This list is checked by the require function to find out if a file needs to be loaded. If that file is already in this list it is not loaded. Typically a file will have (provide 'MYNAME)

at its end so that a call to (require 'MYNAME) will only load MYNAME.scm once.

`rab_diphone_dir`
: The default directory for the rab diphone database.

`server_access_list`
: If non-nil this is the exhaustive list of machines and domains from which clients may access the server. This is a list of REGEXs that client host must match. Remember to add the backslashes before the dots. [see Section 28.3 [Server/client API], page 138]

`server_deny_list`
: If non-nil this is a list of machines which are to be denied access to the server absolutely, irrespective of any other control features. The list is a list of REGEXs that are used to matched the client hostname. This list is checked first, then server_access_list, then passwd. [see Section 28.3 [Server/client API], page 138]

`server_log_file`
: If set to t server log information is printed to standard output of the server process. If set to nil no output is given. If set to anything else the value is used as the name of file to which server log information is appended. Note this value is checked at server start time, there is no way a client may change this. [see Section 28.3 [Server/client API], page 138]

`server_max_clients`
: In server mode, the maximum number of clients supported at any one time. When more that this number of clients attach simulaneous the last ones are denied access. Default value is 10. [see Section 28.3 [Server/client API], page 138]

`server_passwd`
: If non-nil clients must send this passwd to the server followed by a newline before they can get a connection. It would be normal to set this for the particular server task. [see Section 28.3 [Server/client API], page 138]

`server_port`
: In server mode the inet port number the server will wait for connects on. The default value is 1314. [see Section 28.3 [Server/client API], page 138]

`sgml_parse_progname`
: The name of the program to use to parse SGML files. Typically this is nsgml-1.0 from the sp SGML package. [see Section 10.4 [XML/SGML requirements], page 43]

`sonority_glides`
: List of glides (only good w/ radio_speech)

`sonority_liq`
: List of liquids (only good w/ radio_speech)

`sonority_nas`
: List of nasals (only good w/ radio_speech)

`sonority_v_obst`
: List of voiced obstruents for use in sonority scaling (only good w/ radio_speech)

`sonority_vless_obst`
: List of voiceless obstruents for use in sonority scaling (only good w/ radio_speech)

`SynthTypes`
: List of synthesis types and functions used by the utt.synth function to call appropriate methods for wave synthesis.

`system-voice-path`
: Additional directory not near the load path where voices can be found, this can be redefined in lib/sitevars.scm if desired.

`tilt_accent_list`
: List of events containing accents in tilt model.

`tilt_boundary_list`
: List of events containing boundaries in tilt model.

`tobi_support_yn_questions`
: If set a crude final rise will be added at utterance that are judged to be yesy/no questions. Namely ending in a ? and not starting with a wh-for word.

`token.letter_pos`
: The part of speech tag (valid for your part of speech tagger) for individual letters. When the tokenizer decide to pronounce a token as a list of letters this tag is added to each letter in the list. Note this should be from the part of speech set used in your tagger which may not be the same one that appears in the actual lexical entry (if you map them afterwards). This specifically allows `"a"` to come out as ae rather than `@`.

`token.prepunctuation`
: A string of characters which are to be treated as preceding punctuation when tokenizing text. Prepunctuation symbols will be removed from the text of the token and made available through the `"prepunctuation"` feature. [see Section 15.1 [Tokenizing], page 75]

`token.punctuation`
: A string of characters which are to be treated as punctuation when tokenizing text. Punctuation symbols will be removed from the text of the token and made available through the `"punctuation"` feature. [see Section 15.1 [Tokenizing], page 75]

`token.singlecharsymbols`
: Characters which have always to be split as tokens. This would be usual is standard text, but is useful in parsing some types of file. [see Section 15.1 [Tokenizing], page 75]

`token.unknown_word_name`
: When all else fails and a pronunciation for a word or character can't be found this word will be said instead. If you make this `""` them the unknown word will

simple be omitted. This will only really be called when there is a bug in the lexicon and characters are missing from the lexicon. Note this word should be in the lexicon.

`token.whitespace`
: A string of characters which are to be treated as whitespace when tokenizing text. Whitespace is treated as a separator and removed from the text of a token and made available through the "whitespace" feature. [see Section 15.1 [Tokenizing], page 75]

`token_most_common`
: A list of (English) words which were found to be most common in an text database and are used as discriminators in token analysis.

`token_pos_cart_trees`
: This is a list of pairs or regex plus CART tree. Tokens that match the regex will have the CART tree aplied, setting the result as the token_pos feature on the token. The list is checked in order and only the first match will be applied.

`tts_hooks`
: Function or list of functions to be called during text to speech. The function tts_file, chunks data into Utterances of type Token and applies this hook to the utterance. This typically contains the utt.synth function and utt.play. [see Chapter 9 [TTS], page 33]

`tts_text_modes`
: An a-list of text modes data for file type specific tts functions. See the manual for an example. [see Section 9.2 [Text modes], page 34]

`UttTypes`
: List of types and functions used by the utt.synth function to call appropriate methods.

`var-docstrings`
: An assoc-list of variable names and their documentation strings.

`voice-location-trace`
: Set t to print voice locations as they are found

`voice-locations`
: Association list recording where voices were found.

`voice-path`
: List of places to look for voices. If not set it is initialised from load-path by appending "voices/" to each directory with system-voice-path appended.

`voice_default`
: A variable whose value is a function name that is called on start up to the default voice. [see Section 6.3 [Site initialization], page 19]

`Internal variable containing list of voice descriptions as`
: decribed by proclaim_voice.

`xml_dtd_dir`
: The directory holding standard DTD form the xml parser.

`xxml_elements`
: List of Scheme actions to perform on finding xxML tags.

`xxml_hooks`
: Function or list of functions to be applied to an utterance when parsed with xxML, before tts_hooks.

`xxml_token_hooks`
: Functions to apply to each token.

`xxml_word_features`
: An assoc list of features to be added to the current word when in xxml parse mode.

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.

`(%%closure ENVIRONMENT CODE)`
: Make a closure from given environment and code.

`(%%closure-code CLOSURE)`
: Return code part of closure.

`(%%closure-env CLOSURE)`
: Return environment part of closure.

`(%%stack-limit AMOUNT SILENT)`
: Set stacksize to AMOUNT, if SILENT is non nil do it silently.

`(* NUM1 NUM2 ...)`
: Returns the product of NUM1 and NUM2 ... An error is given is any argument is not a number.

`(*catch TAG . BODY)`
: Evaluate BODY, if a *throw occurs with TAG then return value specified by *throw.

`(*throw TAG VALUE)`
: Jump to *catch with TAG, causing *catch to return VALUE.

`(+ NUM1 NUM2 ...)`
: Returns the sum of NUM1 and NUM2 ... An error is given is any argument is not a number.

`(- NUM1 NUM2)`
: Returns the difference between NUM1 and NUM2. An error is given is any argument is not a number.

`(/ NUM1 NUM2)`
: Returns the quotient of NUM1 and NUM2. An error is given is any argument is not a number.

`(:backtrace [FRAME])`
: This function called *immediately* after an error will display a backtrace of the functions evaluated before the error. With no arguments it lists all stack frames, with the (possibly shortened) forms that were evaluated at that level. With a numeric argument it displays the form at that level in full. This function only works at top level in the read-eval-print loop (command interpreter). Note that any valid command will leave the backtrace stack empty. Also note that backtrace itself does not reset the backtrace, unless you make an error in calling it.

`(< NUM1 NUM2)`
: Returns t if NUM1 is less than NUM2, nil otherwise. An error is given is either argument is not a number.

`(<= NUM1 NUM2)`
: Returns t if NUM1 is less than or equal to NUM2, nil otherwise. An error is given is either argument is not a number.

`(> NUM1 NUM2)`
: Returns t if NUM1 is greater than NUM2, nil otherwise. An error is given is either argument is not a number.

`(>= NUM1 NUM2)`
: Returns t if NUM1 is greater than or equal to NUM2, nil otherwise. An error is given is either argument is not a number.

`(acost:build_disttabs UTTTYPES PARAMS)`
: Built matrices of distances between each ling_item in each each list of ling_items in uttypes. Uses acoustic weights in PARAMS and save the result as a matrix for later use.

`(acost:file_difference FILENAME1 FILENAME2 PARAMS)`
: Load in the two named tracks and find the acoustic difference over all based on the weights in PARAMS.

`(acost:utt.load_coeffs UTT PARAMS)`
: Load in the acoustic coefficients into UTT and set the Acoustic_Coeffs feature for each segment in UTT.

`(add-doc-var VARNAME DOCSTRING)`
: Add document string DOCSTRING to VARNAME. If DOCSTRING is nil this has no effect. If VARNAME already has a document string replace it with DOCSTRING.

`(and CONJ1 CONJ2 ... CONJN)`
: Evaluate each conjunction CONJn in turn until one evaluates to nil. Otherwise return value of CONJN.

`(append L0 L1 ...)`
: Append each list to the first list in turn.

`(append LIST1 LIST2)`
: Returns LIST2 appended to LIST1, LIST1 is distroyed.

`(apply FUNC ARGS)`
: Call FUNC with ARGS as arguments.

`(apply_hooks HOOK OBJ)`
: Apply HOOK(s) to OBJ. HOOK is a function or list of functions that take one argument.

`(apply_method METHOD UTT)`
: Apply the appropriate function to utt defined in parameter.

`(approx-equal? a b diff)`
: True is the difference between a b is less than diff. This allows equality between floats which may have been written out and read in and hence have slightly different precision.

`(aref ARRAY INDEX)`
: Return ARRAY[INDEX]

`(aset ARRAY INDEX VAL)`
: Set ARRAY[INDEX] = VAL

`(assoc KEY A-LIST)`
: Return pair with KEY in A-LIST or nil.

`(assoc_string key alist)`
: Look up key in alist using string-equal. This allow indexing by string rather than just symbols.

`(assq ITEM ALIST)`
: Returns pairs from ALIST whose car is ITEM or nil if ITEM is not in ALIST.

`(atom X)`
: True if X is not a cons cells, nil otherwise.

`(audio_mode MODE)`
: Control audio specific modes. Five subcommands are supported. If MODE is async, start the audio spooler so that Festival need not wait for a waveform to complete playing before continuing. If MODE is sync wait for the audio spooler to empty, if running, and they cause future plays to wait for the playing to complete before continuing. Other MODEs are, close which waits for the audio spooler to finish any waveforms in the queue and then closes the spooler (it will restart on the next play), shutup, stops the current waveform playing and empties the queue, and query which lists the files in the queue. The queue may be up to five waveforms long. [see Chapter 23 [Audio output], page 103]

`(backquote FORM)`
: Backquote function for expanding forms in macros.

`(basename PATH SUFFIX)`
: Return a string with directory removed from basename. If SUFFIX is specified remove that from end of PATH. Basically the same function as the UNIX command of the same name.

`(begin . BODY)`
: Evaluate s-expressions in BODY returning value of from last expression.

`(english_token_to_words TOKENSTREAM TOKENNAME)`
: Returns a list of words expanded from TOKENNAME. Note that as this function may be called recursively TOKENNAME may not be the name of TOKENSTREAM.

`(Builtin_PostLex UTT)`
: Post-lexical rules. Currently only vowel reduction applied to each syllable using postlex_vowel_reduce_cart_tree, and the table of vowel reduction pairs in postlex_vowel_reduce_table.

`(caar X)` Return the (car (car X)).

`(caddr X)` Return the (car (cdr (cdr X))).

`(cadr X)` Return the (car (cdr X)).

`(car DATA1)`
Returns car of DATA1. If DATA1 is nil or a symbol, return nil.

`(cd DIRNAME)`
Change directory to DIRNAME, if DIRNAME is nil or not specified change directory to user's HOME directory.

`(cdar X)` Return the (cdr (car X)).

`(cdddr X)` Return the (cdr (cdr (cdr X))).

`(cddr X)` Return the (cdr (cdr X)).

`(cdr DATA1)`
Returns cdr of DATA1. If DATA1 is nil or a symbol, return nil.

`(Pauses UTT)`
Predict pause insertion.

`(Classic_Phrasify UTT)`
Creates phrases from words, if pos_supported is non-nil, a more elaborate system of prediction is used. Here probability models based on part of speech and B/NB distribution are used to predict breaks. This system uses standard Viterbi decoding techniques. If pos_supported is nil, a simple CART-based prediction model is used. [see Chapter 17 [Phrase breaks], page 81]

`(Classic_POS UTT)`
Predict part of speech tags for the existing word stream. If the variable pos_lex_name is nil nothing happens, otherwise it is assumed to point to a lexicon file giving part of speech distribution for words. An ngram model file should be in pos_ngram_name. The system uses standard Viterbi decoding techniques. [see Chapter 16 [POS tagging], page 79]

`(Classic_PostLex utt)`
Apply post lexical rules (both builtin and those specified in postlex_rules_hooks).

`(Classic_Word UTT)`
Build the syllable/segment/SylStructure from the given words using the Lexicon. Uses part of speech information in the lexicon look up if present.

`(clunits:load_db PARAMS)`
Load index file for cluster database and set up params.

`(Clunits_Get_Units UTT)`
Construct Unit relation from the selected units in Segment and extract their parameters from the clunit db.

`(Clunits_Select UTT)`
Select units from current databases using cluster selection method.

`(Clunits_Simple_Wave UTT)`
: Naively concatenate signals together into a single wave (for debugging).

`(Clunits_Windowed_Wave UTT)`
: Use hamming window over edges of units to join them, no prosodic modification though.

`(compile-file FILENAME)`
: Compile lisp forms in FILENAME.scm to FILENAME.bin.

`(compile_library)`
: Compile all the scheme files in the library directory.

`(cons DATA1 DATA2)`
: Construct cons pair whose car is DATA1 and cdr is DATA2.

`(cons-array DIM KIND)`
: Construct array of size DIM and type KIND. Where KIND may be one of double, long, string or lisp.

`(copy-list LIST)`
: Return new list with same members as LIST.

`(debug_output ARG)`
: If ARG is non-nil cause all future debug output to be sent to cerr, otherwise discard it (send it to /dev/null).

`(def_feature_docstring FEATURENAME FEATUREDOC)`
: As some feature are used directly of stream items with no accompanying feature function, the features are just values on the feature list. This function also those features to have an accompanying documentation string.

`(define (FUNCNAME ARG1 ARG2 ...) . BODY)`
: Define a new function call FUNCNAME with arguments ARG1, ARG2 ... and BODY.

`(defmac-macro MACRONAME FORM)`
: Define a macro. Macro expand FORM in-line.

`(defPhoneSet PHONESETNAME FEATURES PHONEDEFS)`
: Define a new phoneset named PHONESETNAME. Each phone is described with a set of features as described in FEATURES. Some of these FEATURES may be significant in various parts of the system. Copying an existing description is a good start. [see Chapter 12 [Phonesets], page 47]

`(defSynthType TYPE . BODY)`
: Define a new wave synthesis type. TYPE is an atomic type that identifies the type of synthesis. BODY is evaluated with argument utt, when utt.synth is called with an utterance of type TYPE. [see Section 14.2 [Utterance types], page 65]

`(defUttType TYPE . BODY)`
: Define a new utterance type. TYPE is an atomic type that is specified as the first argument to the function Utterance. BODY is evaluated with argument

utt, when utt.synth is called with an utterance of type TYPE. You almost always require the function Initialize first. [see Section 14.2 [Utterance types], page 65]

`(delete-file FILENAME)`
Delete named file.

`(delq ITEM LIST)`
Destructively delete ITEM from LIST, returns LIST, if ITEM is not first in LIST, cdr of LIST otherwise. If ITEM is not in LIST, LIST is returned unchanged.

`(describe_all_modules)`
Print descriptions of all proclaimed modules

`(describe_module MOD)`
Describe the module named by the symbol MOD.

`(Diphone.group DB_NAME GROUPFILE)`
Create a group file for DB_NAME in GROUPFILE. A group file is a saved image of the database containing only the necessary information. It is an efficient way for loading and using diphone databases and is the recommended format for diphone databases for normal use. [see Section 21.3 [Group files], page 97]

`(Diphone.list)`
List the names of the currently loaded diphone databases.

`(diphone.oc TABLE WEIGHTS)`
For a table of units (diphones) find the optimal coupling points. WEIGHTS is a list of weights to apply to the vector coefficients used in measuring the goodness of the joins.

`(Diphone.select DB_NAME)`
Select a preloaded diphone set named DB_NAME, diphone sets are identified by their name parameter.

`(Diphone_Init PARAMS)`
Initialize a general diphone database. PARAMS are an assoc list of parameter name and value. [see Section 21.4 [Diphone_Init], page 97]

`(Diphone_Synthesize UTT)`
Synthesize a waveform using the currently selected diphone database. This is called from Synthesize when the Synth_Method Parameter has the value Diphone. [see Chapter 21 [Diphone synthesizer], page 95]

`(directory-entries DIRECTORY &opt NOFLAGDIR)`
Return a list of the entries in the directory. If NOFLAGDIR is non-null don't check to see which are directories.

`(display utt)`
Display an utterance's waveform, F0 and segment labels in Xwaves. Xwaves must be running on the current machine, with a labeller for this to work.

`(doc SYMBOL)`
: Return documentation for SYMBOL.

`(Donovan_Init PARAMS)`
: Initialize the Donovan LPC diphone database. PARAMS are an assoc list of parameter name and value. The two parameters are index_file (value is a pathname for "diphlocs.txt") and diphone_file (value is a pathname for "lpcdiphs.bin"). [see Section 22.1 [LPC diphone synthesizer], page 101]

`(Donovan_Synthesize UTT)`
: Synthesize a waveform using the Donovan LPC diphone synthesizer. This is called from Synthesize when the Synth_Method Parameter has the value Donovan. [see Section 22.1 [LPC diphone synthesizer], page 101]

`(downcase SYMBOL)`
: Returns a string with the downcased version of SYMBOL's printname.

`(Duration utt)`
: Predict segmental durations using Duration_Method defined in Parameters. Four methods are currently available: averages, Klatt rules, CART tree based, and fixed duration.

`(Duration_Averages UTT)`
: Label all segments with their average duration found from the assoc list of phone names to averages in phoneme_durations. This module is called through the module Duration when the Parameter Duration_Method is set to Averages. [see Section 19.2 [Average durations], page 87]

`(Duration_Default UTT)`
: Label all segments with a fixed duration of 100ms. This module is called through the module Duration when the Parameter Duration_Method is unset or set to Default. [see Section 19.1 [Default durations], page 87]

`(duration_find_stretch utt seg)`
: Find any relavant duration stretch.

`(Duration_Klatt UTT)`
: This uses an implementation of the Klatt Duration rules to predict durations for each segment in UTT. It uses the information in duration_klatt_params for mean and lower bound for each phone. This module is called through the module Duration when the Parameter Duration_Method is set to Klatt. This method modifies its predicted durations by the factor set in the Parameter Duration_Stretch (if set). [see Section 19.3 [Klatt durations], page 88]

`(Duration_LogZScores utt)`
: Predicts duration to segments using the CART tree in duration_logzscore_tree and duration_logzscore_tree_silence which produces a zscore of the log duration. The variable duration_logzscore_ph_info contains (log) means and std for each phone in the set.

`(Duration_Tree UTT)`
: Uses the CART tree in duration_cart_tree to predict absolute durations for each segment in UTT. This module is called through the module Duration when the

Parameter Duration_Method is set to Tree. This method modifies its predicted durations by the factor set in the Parameter Duration_Stretch (if set). [see Section 19.4 [CART durations], page 88]

`(Duration_Tree_ZScores UTT)`
: Uses the CART tree in duration_cart_tree to predict z scores duration values for each segment in UTT. The z scores are converted back to absolute values by the assoc list of phones to means and standard deviations in the variable duration_ph_info. This module is called through the module Duration when the Parameter Duration_Method is set to Tree_ZScores. This method modifies its predicted durations by the factor set in the Parameter Duration_Stretch (if set). [see Section 19.4 [CART durations], page 88]

`(duration_unzscore phname zscore table)`
: Look up phname in table and convert xscore back to absolute domain.

`(email_exit_func)`
: Called on exit email text mode.

`(email_init_func)`
: Called on starting email text mode.

`(email_token_to_words utt token name)`
: Email spcific token to word rules.

`(english_token_to_words TOKEN NAME)`
: Returns a list of words for NAME from TOKEN. This allows the user to customize various non-local, multi-word, context dependent translations of tokens into words. If this function is unset only the builtin translation rules are used, if this is set the builtin rules are not used unless explicitly called. [see Section 15.2 [Token to word rules], page 75]

`(env-lookup VARNAME ENVIRONMENT)`
: Return value of VARNAME in ENVIRONMENT.

`(eof_val)`
: Returns symbol used to indicate end of file. May be used (with eq?) to determine when end of file occurs while reading files.

`(eq? DATA1 DATA2)`
: Returns t if DATA1 and DATA2 are the same object.

`(equal? A B)`
: t if s-expressions A and B are recursively equal, nil otherwise.

`(eqv? DATA1 DATA2)`
: Returns t if DATA1 and DATA2 are the same object or equal numbers.

`(error MESSAGE DATA)`
: Prints MESSAGE about DATA and throws an error.

`(eval DATA)`
: Evaluate DATA and return result.

`(exit [RCODE])`
: Exit from program, if RCODE is given it is given as an argument to the system call exit.

`(exp NUM)`
: Return e**NUM.

`(extract_tokens FILE TOKENS OUTFILE)`
: Find all occurences of TOKENS in FILE and output specified context around the token. Results are appended to OUTFILE, if OUTFILE is nil, output goes to stdout.

`(f2b_lts WORD FEATURES)`
: Letter to sound rule system for f2b (American English), uses the NRL LTS ruleset and maps the result to the radio phone set.

`(fasdump FILENAME FORMS)`
: Fast dump FORMS into FILENAME.

`(fasl-close TABLE)`
: Close fasl table.

`(fasl-open FILENAME MODE)`
: Open fasl FILENAME as MODE. Returns a fasl-table.

`(fasload FILENAME ARGS)`
: Fast load FILENAME.

`(fasload_library FILENAME)`
: Load binary file from library

`(fast-print P TABLE)`

`(fast-read TABLE)`

`(fclose FILEP)`
: Close filepoint FILEP.

`(feats.get FEATS FEATNAME)`
: Return value of FEATNAME (which may be a simple feature name or a pathname) in FEATS. If FEATS is nil a new feature set is created

`(feats.make)`
: Return an new empty features object.

`(feats.present FEATS FEATNAME)`
: Return t is FEATNAME is present in FEATS, nil otherwise.

`(feats.remove FEATS FEATNAME)`
: Remove feature names FEATNAME from FEATS.

`(feats.set FEATS FEATNAME VALUE)`
: Set FEATNAME to VALUE in FEATS.

`(feats.tolisp FEATS)`
: Gives a lisp representation of the features, this is a debug function and may or may not exist tomorrow.

`(feats.value_sort FEATURES NAME)`

`(festival_warranty)`
: Display Festival's copyright and warranty. [see Chapter 2 [Copying], page 3]

`(fflush FILEP)`
: Flush FILEP. If FILEP is nil, then flush stdout.

`(find_month_from_number token string-number)`
: Find the textual representation of the month from the given string number

`find_peak_seg_anchor ie pk_pos`
: Part of the workings of peak_segment_anchor.

`(Fixed_Prosody UTT)`
: Add fixed duration and fixed monotone F0 to the sgements in UTT. Uses values of FP_duration and FP_F0 as fixed values.

`(flatten LIST)`
: Return flatend list (list of all atoms in LIST).

`(fopen FILENAME HOW)`
: Return file pointer for FILENAME opened in mode HOW.

`(format FD FORMATSTRING ARG0 ARG1 ...)`
: Output ARGs to FD using FROMATSTRING. FORMATSTRING is like a printf formatstrng. FD may be a filedescriptor, or t (standard output) or nil (return as a string). Note not all printf format directive are supported. %l is additionally support for Lisp objects. [see Section 8.4 [Scheme I/O], page 31]

`(fread BUFFER FILE)`
: BUFFER is a string of length N, N bytes are read from FILE into BUFFER.

`(fringe_command SERVER PACKAGE OPERATION ARGS)`
: Send command to the fringe server SERVER. ARGS should be an association list of key-value pairs.

`(fringe_command_string SERVER COMMAND)`
: Send COMMAND to the fringe server SERVER.

`(fringe_connect SERVER)`
: Re-open the connection to the server.

`(fringe_disconnect SERVER)`
: Close the connection to the server.

`(fringe_read_server_table &opt FILENAME)`
: Read the users table of fringe servers, or the table in FILENAME if given.

`(fringe_server &opt NAME)`
: Return a connection to a fringe server with the given name. If name is omitted it defaults to "fringe".

`(fringe_servers)`
: Returns a list of the know fringe servers. This doesn't guarantee that they are still running.

`(fseek FILEP OFFSET DIRECTION)`
: Position FILEP to OFFSET. If DIRECTION is 0 offset is from start of file. If DIRECTION is 1, offset is from current position. If DIRECTION is 2 offset is from end of file.

`(ftell FILEP)`
: Returns position in file FILEP is currently pointing at.

`(fwarning MODE)`
: For controlling various levels of warning messages. If MODE is nil, or not specified stop all warning messages from being displayed. If MODE display warning messages.

`(fwrite BUFFER FILE)`
: Write BUFFER into FILE.

`(gc)` Collect garbage now, where gc method supports it.

`(gc-status OPTION)`
: Control summary information during garbage collection. If OPTION is t, output information at each garbage collection, if nil do gc silently.

`(Gen_Viterbi UTT)`
: Applies viterbi search algorithm based on the parameters in gen_vit_params. Basically allows user candidate selection function combined with ngrams.

`(get SYM KEY)`
: Get property named KEY for SYM.

`Gets the c/v value of the segment within a syllable.`

`(get_param name params default)`
: Get named parameters in params returning default if its not present.

`(get_url URL OUTFILE)`
: Get URL and put contents in OUTFILE. Currently only http, and file type URLs are supported.

`(getc FILEP)`
: Get next character from FILEP. Character is returned as a number. If FILEP is nil, or not specified input comes from stdin.

`(getenv VARNAME)`
: Returns value of UNIX environment variable VARNAME, or nil if VARNAME is unset.

`(getpid)` Return process id.

`(href TABLE KEY)`
: Return value in hash table TABLE with KEY.

`(hset TABLE KEY VALUE)`
: Set hash table TABLE KEY to VALUE.

`(if COND TRUEPART FALSEPART)`
: If COND evaluates to non-nil evaluate TRUEPART and return result, otherwise evaluate and return FALSEPART. If COND is nil and FALSEPART is nil, nil is returned.

`(Initialize UTT)`
: This module should be called first on all utterances it does some necessary initialization of the utterance and loads the base streams with the information from the input form.

`(insert_initial_pause UTT)`
: Always have an initial silence if the utterance is non-empty. Insert a silence segment after the last segment in WORDITEM in UTT.

`(insert_pause UTT WORDITEM)`
: Insert a silence segment after the last segment in WORDITEM in UTT.

`(Int_Targets utt)`
: The second stage in F0 prediction. This generates F0 targets related to segments using one of three methods, a simple hat, linear regression based on ToBI markings, and a simple declining slope. This second part deals with actual F0 values and durations, while the previous section only deals with accent (and boundary tone) assignment. [see Chapter 18 [Intonation], page 83]

`(Int_Targets_Default UTT)`
: This module creates two Targets causing a simple downward continuous F0 through the whole utterance. The code is in an appropriate named file called duffint. This module is called when the Parameter Int_Method is not set or set to Default. This module is called through the Int_Targets module. Optional parameters for a start value (default 130) and end value (default 110) may be set in the variable diffint_params. This can be used to generate a monotone intonation with a setting like (set! duffint_params '((start 100) (end 100))). [see Section 18.1 [Default intonation], page 83]

`(Int_Targets_General UTT)`
: Add targets based on the functions defined in int_general_params. This method allows quite detailed control over the general of targets per syllable, see manual for details and examples. This module is called when the Parameter Int_Method is set to General. This module is called through the Int_Targets module. [see Section 18.5 [General intonation], page 84]

`(Int_Targets_LR UTT)`
: Predict Target F0 points using linear regression from factors such as accent, tone, stress, position in phrase etc. This utterance module is called through the module Int_Targets when the Parameter Int_Method is set to ToBI, even though this technique is not restricted to the ToBI labelling system. [see Section 18.3 [Tree intonation], page 84]

`(Int_Targets_Relation UTT)`

`(Int_Targets_Simple UTT)`
: Naively add targets for hat shaped accents for each accent in the IntEvent stream. This module is called when the Parameter Int_Method is set to Simple. This module is called through the Int_Targets module. [see Section 18.2 [Simple intonation], page 83]

`(Int_Targets_Tilt utt)`
: Assign Tilt parameters to each IntEvent and then generate the F0 contour and assign targets.

`(intern ATOM)`
: Intern ATOM on the oblist.

`(Intonation utt)`
: Select between different intonation modules depending on the Parameter Int_Method. Currently offers three types: Simple, hats on each content word; ToBI, a tree method for predicting ToBI accents; and Default a really bad method with a simple downward sloping F0. This is the first of a two-stage intonation prediction process. This adds accent-like features to syllables, the second, Int_Targets generates the F0 contour itself. [see Chapter 18 [Intonation], page 83]

`(Intonation_Default UTT)`
: this method is such a bad intonation module that it does nothing at all. This utterance module is called when the Parameter Int_Method is not set or set to Default. This module is called through the Intonation module. [see Section 18.1 [Default intonation], page 83]

`(Intonation_Simple)`
: Assign accents to each content word, creating an IntEvent stream. This utterance module is called when the Parameter Int_Method is set to Simple. This module is called through the Intonation module. [see Section 18.2 [Simple intonation], page 83]

`(Intonation_Tilt utt)`
: Assign accent and boundary IntEvents to each syllable, and fill in spaces with silence and connections.

`(Intonation_Tree UTT)`
: Use the CART trees in int_tone_cart_tree and int_accent_cart_tree to create an IntEvent stream of tones and accents related to syllables. This module is called through the Intonation module and is selected when the Parameter Int_Method is ToBI. [see Section 18.3 [Tree intonation], page 84]

`(intro)`
: Synthesize an introduction to the Festival Speech Synthesis System.

`(intro-spanish)`
: Synthesize an introduction to the Festival Speech Synthesis System in spanish. Spanish voice must already be selected for this.

`(item.add_link ITEMFROM ITEMTO)`
: Add a link from ITEMFROM to ITEMTO is the relation ITEMFROM is in.

`(item.append_daughter ITEM1 ITEM2)`
: Add a ITEM2 a new daughter (right-most) to ITEM1 in the relation of ITEM1. If ITEM2 is of type item then it is added directly otherwise ITEM2 is treated as a description of an item and a one is created with that description (name features).

`(item.daughter1 ITEM)`
: Return the first daughter of ITEM, or nil if there is none.

`(item.daughter1_to s relname)`
: Follow daughter1 links of s in its current relation until an item is found that is also in relname, is s is in relname it is returned. The return item is returned in relation relname, or nil if there is nothing in relname.

`(item.daughter2 ITEM)`
: Return the second daughter of ITEM, or nil if there is none.

`(item.daughtern ITEM)`
: Return the last daughter of ITEM, or nil if there is none.

`(item.daughter1_to s relname)`
: Follow daughtern links of s in its current relation until an item is found that is also in relname, is s is in relname it is returned. The return item is returned in relation relname, or nil if there is nothing in relname.

`(item.daughters parent)`
: Return a list of all daughters of parent.

`(item.delete ITEM)`
: Remove this item from all relations it is in and delete it.

`(item.down ITEM)`
: Return the item below ITEM, or nil if there is none.

`(item.exchange_tree FROM TO)`
: Exchanged contents of FROM and TO, and descendents of FROM and TO. Returns t if successful, or nil if FROM or TO contain each other.

`(item.feat ITEM FEATNAME)`
: Return value of FEATNAME (which may be a simple feature name or a pathname) of ITEM.

`(item.features ITEM EVALUATE_FEATURES))`
: Returns all features in ITEM as an assoc list.

`(item.first_leaf ITEM)`
: Returns he left most leaf in the tree dominated by ITEM. This is like calling item.daughter1 recursively until an item with no daughters is found.

`(item.get_utt ITEM)`
: Get utterance from given ITEM (if possible).

`(item.insert ITEM1 ITEM2 DIRECTION)`
: Insert ITEM2 in ITEM1's relation with repsect to DIRECTION. If DIRECTION is unspecified, after, is assumed. Valid DIRECTIONS as before, after,

above and below. Use the functions item.insert_parent and item.append_daughter for specific tree adjoining. If ITEM2 is of type item then it is added directly, otherwise it is treated as a description of an item and new one is created.

`(item.insert_parent ITEM1 ITEM2)`
: Insert a new parent between this ITEM1 and its parentm in ITEM1's relation. If ITEM2 is of type item then it is added directly, otherwise it is treated as a description of an item and one is created with that description (name features).

`(item.last_leaf ITEM)`
: Returns he right most leaf in the tree dominated by ITEM. This is like calling item.daughtern recursively until an item with no daughters is found.

`(item.relation.leafs item relname)`
: Return a list of the leafs of this item in this relation.

`(item.link1 ITEM)`
: Return first item linked to ITEM in current relation.

`(item.link2 ITEM)`
: Return second item linked to ITEM in current relation.

`(item.linkedfrom ITEM)`
: Return the item tht is linked to ITEM.

`(item.linkn ITEM)`
: Return last item linked to ITEM in current relation.

`(item.merge FROM TO)`
: Merge FROM into TO making them the same items. All features in FROM are merged into TO and all references to FROM are made to point to TO.

`(item.move_tree FROM TO)`
: Move contents, and descendents of FROM to TO. Old daughters of TO are deleted. FROM will be deleted too if it is being viewed as the same same relation as TO. FROM will be deleted from its current place in TO's relation. Returns t if successful, returns nil if TO is within FROM.

`(item.name ITEM)`
: Returns the name of ITEM. [see Section 14.5 [Accessing an utterance], page 68]

`(item.next ITEM)`
: Return the next ITEM in the current relation, or nil if there is no next.

`(item.next_item ITEM)`
: Will give next item in this relation visiting every item in the relation until the end. Traverses in pre-order, root followed by daughters (then siblings).

`(item.next_leaf ITEM)`
: Return the next leaf item (i.e. one with no daughters) in this relation. Note this may traverse up and down the relation tree significantly to find it.

`(item.next_link ITEM)`
: Return next item licked to the same item ITEM is linked to.

`(item.parent ITEM)`
: Return the item of ITEM, or nil if there is none.

`(item.parent_to s relname)`
: Find the first ancestor of s in its current relation that is also in relname. s is treated as an ancestor of itself so if s is in relname it is returned. The returned value is in will be in relation relname or nil if there isn't one.

`(item.prepend_daughter ITEM1 ITEM2)`
: Add a ITEM2 a new daughter (left-most) to ITEM1 in the relation of ITEM1. If ITEM2 is of type item then it is added directly otherwise ITEM2 is treated as a description of an item and a one is created with that description (name features).

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

`(item.raw_feat ITEM FEATNAME)`
: Return value of FEATNAME as native features structure (which may be a simple feature name or a pathname) of ITEM.

`(item.relation ITEM RELATIONNAME)`
: Return the item such whose relation is RELATIONNAME. If ITEM is not in RELATIONNAME then nil is return.

`(item.relation.append_daughter parent relname daughter)`
: Make add daughter to parent as a new daughter in relname.

`(item.relation.daughter1 item relname)`
: Return the first daughter of this item in this relation.

`(item.relation.daughter2 item relname)`
: Return the second daughter of this item in this relation.

`(item.relation.daughtern item relname)`
: Return the final daughter of this item in this relation.

`(item.relation.daughters parent relname)`
: Return a list of all daughters of parent by relname.

`(item.relation.first item relname)`
: Return the most previous item from this item in this relation.

`(item.relation.insert si relname newsi direction)`
: Insert newsi in relation relname with respect to direction. If direction is ommited after is assumed, valid directions are after before, above and below. Note you should use item.relation.append_daughter for tree adjoining. newsi maybe a item itself of a LISP description of one.

`(item.relation.leafs item relname)`
: Return a list of the leafs of this item in this relation.

`(item.relationname ITEM)`
: Return the name of the relation this ITEM is currently being viewed through.

`(item.relation.next item relname)`
: Return the next item in this relation.

`(item.relation.parent item relname)`
: Return the parent of this item in this relation.

`(item.relation.prev item relname)`
: Return the previous item in this relation.

`(item.relation.remove ITEM RELATIONNAME)`
: Remove this item from Relation, if it apears in no other relation it will be deleted too, in contrast item.delete will remove an item from all other relations, while this just removes it from this relation. Note this will also remove all daughters of this item in this relation from this relation.

`(item.relationnames ITEM)`
: Return a list of names of the relations this item is in.

`(item.remove_feature ITEM FNAME)`
: Remove feature named FNAME from ITEM. Returns t is successfully remove, nil if not found.

`(item.root s)`
: Follow parent link until s has no parent.

`(item.set_feat ITEM FEATNAME VALUE)`
: Set FEATNAME to VALUE in ITEM.

`(item.set_function ITEM FEATNAME FEATFUNCNAME)`
: Set FEATNAME to feature function name FEATFUNCNAME in ITEM.

`(item.set_name ITEM NAME)`
: Sets ITEM's name to NAME. [see Section 14.5 [Accessing an utterance], page 68]

`(item.up ITEM)`
: Return the item above ITEM, or nil if there is none.

`(lambda (ARG1 ARG2 ...) . BODY)`
: Create closure (anonymous function) with arguments ARG1, ARG2 ... and BODY.

`(language_american_english)`
: Set up language parameters for Aemerican English.

`(language_british_english)`
: Set up language parameters for British English.

`(language_spanish)`
: Set up language parameters for Castillian Spanish.

`(language_british_english)`
: Set up language parameters for British English.

`(language_scots_gaelic)`
: Set up language parameters for Scots Gaelic.

`(language_welsh)`
Set up language parameters for Welsh.

`(last A)` Last (cdr) element in list A.

`(lastline STRING)`
Returns the part of the string which between the last newline and the end of string.

`(length LIST)`
Return length of LIST, or 0 if LIST is not a list.

`(let-internal STUFF)`
Internal function used to implement let.

`(let ((VAR1 VAL1) (VAR2 VAL2) ...) . BODY)`
Evaluate BODY in an environment where VAR1 is set to VAL1, VAR2 is set to VAL2 etc.

`(lex.add.entry ENTRY)`
Add ENTRY to the addenda of the current lexicon. As the addenda is checked before the compiled lexicon or letter to sound rules, this will cause ENTRY to be found before all others. If a word already in the addenda is added again the most recent addition will be found (part of speech tags are respected in the look up). [see Section 13.1 [Lexical entries], page 49]

`(lex.compile ENTRYFILE COMPILEFILE)`
Compile the list of lexical entries in ENTRYFILE into a compiled file in COMPILEFILE. [see Section 13.2 [Defining lexicons], page 50]

`(lex.create LEXNAME)`
Create a new lexicon of name LEXNAME. If it already exists, the old one is deleted first. [see Section 13.2 [Defining lexicons], page 50]

`(lex.entrycount WORD)`
Return the number of entries in the compiled lexicon that match this word. This is used in detecting homographs.

`(lex.list)`
List names of all currently defined lexicons.

`(lex.lookup WORD FEATURES)`
Lookup word in current lexicon. The addenda is checked first, if WORD with matching FEATURES (so far this is only the part of speech tag) is not found the compiled lexicon is checked. Only if the word is still not found the letter to sound rules (or whatever method specified by the current lexicon's lts.method is used). [see Section 13.3 [Lookup process], page 51]

`(lex.lookup_all WORD)`
Return list of all entries in the addenda and compiled lexicon that match this word. The letter to sound rules and user defined unknown word function is ignored.

`(lex.select LEXNAME)`
Select LEXNAME as current lexicon. The name of the previously selected lexicon is returned.

`(lex.set.compile.file COMPFILENAME)`
Set the current lexicon's compile file to COMPFILENAME. COMPFILENAME is a compiled lexicon file created by lex.compile. [see Section 13.2 [Defining lexicons], page 50]

`(lex.set.lts.method METHOD)`
Set the current lexicon's letter-to-sound method to METHOD. METHOD can take any of the following values: Error (the default) signal a festival error if a word is not found in the lexicon; lts_rules use the letter to sound rule set named by lts_ruleset; none return simply nil in the pronunciation field; function use call the two argument function lex_user_unknown_word (as set by the user) with the word and features to provide an entry. [see Section 13.4 [Letter to sound rules], page 52]

`(lex.set.lts.ruleset RULESETNAME)`
Set the current lexicon's letter-to-sound ruleset to RULESETNAME. A ruleset of that name must already be defined. This is used if lts.method is set to lts_rules. [see Section 13.4 [Letter to sound rules], page 52]

`(lex.set.phoneset PHONESETNAME)`
Set current lexicon's phone set to PHONESETNAME. PHONESETNAME must be a currently defined (and, of course, loaded) phone set. [see Section 13.2 [Defining lexicons], page 50]

`(lex.set.pos.map POSMAP)`
A reverse assoc-list mapping part of speech tags to the lexical part of speech tag set. [see Section 13.1 [Lexical entries], page 49]

`(lex.set.post_hooks HOOKS)`
Set a function or list of functions that are to be applied to the entry after lookup. Returns previous value [see Section 13.1 [Lexical entries], page 49]

`(lex.set.pre_hooks HOOKS)`
Set a function or list of functions that are to be applied to the entry before lookup. Returns previous value [see Section 13.1 [Lexical entries], page 49]

`(lex.syllabify.phstress PHONELIST)`
Syllabify the given phone list (if current phone set). Vowels may have the numerals 0, 1, or 2 as suffixes, if so these are taken to be stress for the syllable they are in. This format is similar to the entry format in the CMU and BEEP lexicons. [see Section 13.2 [Defining lexicons], page 50]

`(lex_user_unknown_word WORD FEATS)`
Function called by lexicon when 'function type letter to sound rules is defined. It is the user's responsibility to defined this function themselves when they want to deal with unknown words themselves.

`(library_expand_filename FILENAME)`
: Search for filename by appending FILENAME to each member of load-path. Full expanded pathname is returned. If not found in load-path FILENAME is returned.

`(linear_regression ITEM MODEL)`
: Use linear regression MODEL on ITEM. MODEL consists of a list of features, weights and optional map list. E.g. ((Intercept 100) (tobi_accent 10 (H* !H*))).

`(list A0 A1 ...)`
: Return list containing A0 A1 ...

`(load FILENAME OPTION)`
: Load s-expressions in FILENAME. If OPTION is nil or unspecified evaluate each s-expression in FILENAME as it is read, if OPTION is t, return them unevaluated in a list.

`(load_library FILENAME)`
: Load file from library, appends FILENAME to each path in load-path until a valid file is found. If none found loads name itself

`(log NUM)` Return natural log of NUM.

`(lr_predict ITEM LRMODEL)`
: Apply the linear regression model LRMODEL to ITEM in. This returns float value by summing the product of the coeffients and values returned by the specificed features in ITEM. [see Section 25.5 [Linear regression], page 123]

`(lts.apply WORD RULESETNAME)`
: Apply lts ruleset RULESETNAME to word returning result. [see Section 13.4 [Letter to sound rules], page 52]

`(lts.in.alphabet WORD RULESETNAME)`
: Returns t is all characters in symbol word (or items in list WORD) are in the alphabet of letter to sound ruleset name RULESETNAME. nil otherwise. [see Section 13.4 [Letter to sound rules], page 52]

`(lts.list)`
: Return list of all current defined LTS rulesets.

`(lts.ruleset NAME RULES SETS)`
: Define a new set of letter to sound rules. [see Section 13.4 [Letter to sound rules], page 52]

`(make-a-doc FILENAME DOCLIST)`
: Make a texinfo document in FILENAME as a texinfo table, items are from DOCLIST. DOCLIST names which doclist to use, it may be one of 'function, 'features or 'vars.

`(make-doc)`
: Find function and variable document strings and save them in texinfo format to respective files.

`(make-list SIZE VALUE)`
Return list of SIZE with each member VALUE.

`(make_tmp_filename)`
Return name of temporary file.

`(manual SECTION)`
Display SECTION in the manual. SECTION is a string identifying a manual section (it could be an initial substring. If SECTION is nil or unspecifed then the Manual table of contents is displayed. This uses netscape to display the manual page so you must have that (use variable manual-browser to identify it) and the variable manual-url pointing to a copy of the manual. [see Section 7.3 [Getting some help], page 28]

`(manual-sym SYMBOL)`
Display the section in the manual that SYMBOL's docstring has identified as the most relevant. The section is named on the last line of a documentation string with no newlines within it prefixed by "[see " with a "]" just immediately before the end of the documentation string. The manual section name is translated to the section in the HTML version of the manual and a URL is and sent to Netscape for display. [see Section 7.3 [Getting some help], page 28]

`(map_to_relation UTT Source_relation Target_relation new_relation)`
From the F0 contour in F0_relation, create a set of pitchmarks in PM_relation. If END_TIME is not nil, Extra pitchmarks will be created at the default interval up to this point

`(mapcar FUNCTION ARGS [ARGS2])`
Apply FUNCTION to each member of ARGS (and [ARGS2]), returning list of return values.

`(MBROLA_Synth UTT)`
Synthesize using MBROLA as external module. Basically dump the info from this utterance. Call MBROLA and reload the waveform into utt. [see Section 22.2 [MBROLA], page 101]

`(member ITEM LIST)`
Returns subset of LIST whose car is ITEM if it exists, nil otherwise.

`(member_string STRING LIST)`
Returns subset of LIST whose car is STRING if it exists, nil otherwise.

`(module_description MOD)`
Returns the description record of the module named by symbol MOD

`(month_range SC)`
1 if SC's name is > 0 and < 32, 0 otherwise.

`(mrpa_addenda)`
Add a whole host of various entries to the current lexicon with mrpa phones.

`(mt_accent syl)`
Accent or 0 if none.

`(mt_break syl)`
: Break or 0 if none.

`(mt_close n)`
: The number of consituents this is the end of, Effectively the number of closing brackets after this word.

`(mt_fssw s)`
: 1 if first stressed syllable in word, 0 otherwise.

`(mt_lssp s)`
: 1 if last stressed syllable in phrase, 0 otherwise.

`(nfssw s)`
: 1 if second or later stressed syllable in word, 0 otherwise.

`(mt_num_s s)`
: The number of s MetricalValues from here to a w or top.

`(mt_num_w s)`
: The number of w MetricalValues from here to a s or top.

`(mt_open n)`
: The number of consituents this is the start of, Effectively the number of opening brackets before this word.

`(mt_postype syl)`
: Returns single, initial, final or middle.

`(mt_strong s)`
: 1 if all MetricalValues a s to a word, 0 otherwise.

`(MultiProbParse UTT)`
: Parse part of speech tags in Word relation. Unlike ProbParse this allows multiple sentences to appear in the one utterance. The CART tree in eos_tree is used to define end of sentence. Loads the grammar from scfg_grammar_filename and saves the best parse in the Syntax Relation.

`(play_wave FILENAME)`
: Play given wavefile

`(nconc A B)`
: Destructively append B to A, if A is nil return B.

`(ngram.load NAME FILENAME)`
: Load an ngram from FILENAME and store it named NAME for later access.

`(nint NUMBER)`
: Returns nearest int to NUMBER.

`(not DATA)`
: Returns t if DATA is nil, nil otherwise.

`(nth N LIST)`
: Returns nth car of LIST, 0 is car.

`(nth_cdr N LIST)`
: Returns nth cdr of LIST, 0 is LIST.

`(null? DATA)`
: Returns t if DATA is nil, nil otherwise.

`(num_digits SC)`
: Returns number of digits (actually chars) is SC's name.

`Finds the number of postvocalic consonants in a syllable.`

`(number? DATA)`
: Returns t if DATA is a number, nil otherwise.

`(oald_lts_function word feats)`
: Function called for OALD when word is not found in lexicon. Uses LTS rules trained from the original lexicon, and lexical stress prediction rules.

`(oblist)`
: Return oblist.

`(or DISJ1 DISJ2 ...)`
: Evaluate each disjunction DISJn in turn until one evaluates to non-nil. Otherwise return nil.

`(pair? DATA)`
: Returns t if DATA is a cons cell, nil otherwise.

`(Param.def NAME VAL)`
: Set parameter NAME to VAL if not already set

`(Param.get NAME)`
: Get parameter NAME's value (nil if unset)

`(Param.set NAME VAL)`
: Set parameter NAME to VAL (deleting any previous setting)

`(Parameter.def NAME VAL)`
: Set parameter NAME to VAL if not already set. This is an OLD function you shold use Param.def instead.

`(Parameter.get NAME)`
: Get parameter NAME's value (nil if unset). This is an OLD function and may not exist in later versions (or change functionality). This function (unlike Param.get) may return sylbols (rather than strings if the val doesn't contain whitespace (to allow older functions to still work.

`(Parameter.set NAME VAL)`
: Set parameter NAME to VAL (deleting any previous setting). This is an old function and you should use Param.set instead.

`(parse-number SYMBOL)`
: Returns a number form a symbol or string whose print name is a number.

`(parse_url URL)`
: Split URL into a list (protocol host port path) suitable for giving to fopen.

`(path-append DIRECTORY-PATH ADDITION1 ADDITION2 ...)`
: Return a the path for ADDITION in DIRECTORY.

`(path-as-directory PATHNAME)`
: Return PATH as a directory name.

`(path-as-file PATHNAME)`
: Return PATH as a non-directory name.

`(path-basename PATHNAME)`
: Return name part of PATH.

`(path-is-dirname PATHNAME)`
: Is PATH a directory name.

`(path-is-filename PATHNAME)`
: Is PATH a non-directory name.

`(Pauses utt)`
: Insert pauses where required.

`peak_segment_anchor ie`
: Determines what segment acts as the anchor for a peak. Returns number of segments from start of accented syllable to peak.

`peak_wi_seg segment pk_pos`
: Finds if a peak occurs w/i a segment

`(phone_feature phone feat)`
: Return the feature for given phone in current phone set, or 0 if it doesn't exist.

`(Phoneset.description OPTIONS)`
: Returns a lisp for of the current phoneme set. Options is a list of parts of the definition you require. OPTIONS may include, silences, phones, features and/or name. If nil all are returned.

`(Phoneset.list)`
: List the names of all currently defined Phonesets.

`(PhoneSet.select PHONESETNAME)`
: Select PHONESETNAME as current phoneset. [see Chapter 12 [Phonesets], page 47]

`(PhoneSet.silences LIST)`
: Declare LIST of phones as silences. The first in the list should be the "most" silent. [see Chapter 12 [Phonesets], page 47]

`(Phrasify utt)`
: Construct phrasify over Words module.

`(POS utt)`
: Apply part of speech tagging (and possible parsing too) to Word relation.

`(position thing l)`
: What position is thing in l, -1 if it doesn't exist.

`(PostLex utt)`
: Apply post lexical rules to segment stream. These may be almost arbitrary rules as specified by the particular voice, through the postlex_hooks variable. A number of standard post lexical rule sets are provided including reduction, posessives etc. These rules are also used to mark standard segments with their cluster information used in creating diphone names.

`(postlex_apos_s_check UTT)`
: Deal with possesive 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.

`(pow X Y)` Return X**Y.

`(pprintf EXP [FD])`
: Pretty print EXP to FD, if FD is nil print to the screen.

`(print DATA)`
: Print DATA to stdout if textual form. Not a pretty printer.

`(print_string DATA)`
: Returns a string representing the printing of DATA.

`(printfp DATA FILEP)`
: Print DATA to file indicated by file pointer FILEP. File pointers are are created by fopen.

`(probe_file FILENAME)`
: Returns t if FILENAME exists and is readable, nil otherwise.

`(ProbParse UTT)`
: Parse part of speech tags in Word relation. Loads the grammar from scfg_grammar_filename and saves the best parse in the Syntax Relation.

`(proclaim_voice NAME DESCRIPTION)`
: Describe a voice to the systen. NAME should be atomic name, that conventionally will have voice_ prepended to name the basic selection function. OPTIONS is an assoc list of feature and value and must have at least features for language, gender, dialect and description. The first there of these are atomic, while the description is a text string describing the voice.

`(provide FILENAME)`
: Adds FNAME to the variable provided (if not already there). This means that future calls to (require FILENAME) will not cause FILENAME to be re-loaded.

`(putc CHAR FILEP)`
: Put CHAR (a number) as a character to FILEP. If FILEP is nil or not specified output goes to stdout.

`(putprop SYM VAL KEY)`
: Put property VAL named KEY for SYM.

`(puts STRING FILEP)`
: Write STRING (print name of symbol) to FILEP. If FILEP is nil or not specified output goes to stdout.

`(pwd)` Returns current directory as a string.

`(quit)` Exit from program, does not return.

`(quote DATA)`
Return data (unevaluated).

`(rab_diphone_const_clusters UTT)`
Identify consonant clusters, dark ls etc in the segment item ready for diphone resynthesis. This may be called as a post lexical rule through poslex_rule_hooks.

`(rab_diphone_fix_phone_name UTT SEG)`
Add the feature diphone_phone_name to given segment with the appropriate name for constructing a diphone. Basically adds _ if either side is part of the same consonant cluster, adds $ either side if in different syllable for preceding/succeeding vowel syllable, and converts l to ll in coda part of syllables.

`(rab_postlex_syllabics utt)`
Because the lexicon is somewhat random in its used of syllable l n and m this is designed to post process the output inserting schwa before them. Ideally the lexicon should be fixed.

`(rand)` Returns a pseudo random number between 0 and 1 using the libc rand() function.

`(read)` Read next s-expression from stdin and return it.

`(read-from-string SYMBOL)`
Return first s-expression in print name of SYMBOL.

`(readfp FILEP)`
Read and return next s-expression from file indicated by file pointer FILEP. File pointers are created by fopen.

`(remove ITEM LIST)`
(Non-destructively) remove ITEM from LIST.

`(remove-duplicates LIST)`
Remove duplicate items in LIST.

`(remove_leading_zeros name)`
Remove leading zeros from given string.

`(replace BEFORE AFTER)`
Destructively replace contents of cons cell BEFORE with those of AFTER.

`(request FILENAME)`
Checks if FNAME is already provided (member of variable provided) if not tries to loads it, appending ".scm" to FILENAME. Uses load_library to find the file. Unlike require, fname isn't found no error occurs

`(require FILENAME)`
Checks if FNAME is already provided (member of variable provided) if not loads it, appending ".scm" to FILENAME. Uses load_library to find the file.

`(require_module l)`
: Check that certain compile-time modules are included in this installation. l may be a single atom or list of atoms. Each item in l must appear in *modules* otherwise an error is throw.

`(reslpc_resynth FILENAME)`
: Resynthesize FILENAME using LPC plus residual. The current diphone database must have definitions for LPC for this to work.

`(reverse LIST)`
: Returns destructively reversed LIST.

`(replacd A B)`
: Destructively replace the cdr of A with B.

`(save-forms FILENAME FORMS HOW)`
: Save FORMS in FILENAME. If HOW is a appending FORMS to FILENAME, or if HOW is w start from the beginning of FILENAME.

`(save_seg_mbrola_entry ENTRY NAME START DUR TARGS FD)`
: Entry contains, (name duration num_targs start 1st_targ_pos 1st_targ_val).

`(save_segments_mbrola UTT FILENAME)`
: Save segment information in MBROLA format in filename. The format is phone duration (ms) [% position F0 target]*. [see Section 22.2 [MBROLA], page 101]

`(save_waves_during_tts)`
: Save each waveform in the current directory in files "tts_file_XXX.wav". use (save_waves_during_tts_STOP) to stop saving waveforms

`(save_waves_during_tts_STOP)`
: Stop saving waveforms when doing tts.

`(SayPhones PHONES)`
: PHONES is a list of phonemes. This uses the Phones type utterance to synthesize and play the given phones. Fixed duration specified in FP_duration and fixed monotone duration (FP_F0) are used to generate prosody.

`(SayText TEXT)`
: TEXT, a string, is rendered as speech.

`(search-for-voices)`
: Search down voice-path to locate voices.

`(segment_dpitch UTT SEG)`
: Returns delta pitch, this pitch minus previous pitch.

`segs_to_peak sylSyl pk_pos`
: Determines the number of segments from the start of a syllable to an intonation peak

`(send-url-to-netscape URL)`
: Send given URL to netscape for display. This is primarily used to display parts of the manual referenced in documentation strings.

`(set! SYMBOL VAL)`
: Set SYMBOL to have value VAL, returns VAL.

`(set-car! CONS1 DATA1)`
: Set car of CONS1 to be DATA1. Returns CONS1. If CONS1 not of type consp an error is is given. This is a destructive operation.

`(set-cdr! CONS1 DATA1)`
: Set cdr of CONS1 to be DATA1. Returns CONS1. If CONS1 not of type consp an error is is given. This is a destructive operation.

`(set-symbol-value! SYMBOLNAME VALUE)`
: Set SYMBOLNAME's value to VALUE, this is much faster than set! but use with caution.

`(set_backtrace arg)`
: If arg is non-nil a backtrace will be display automatically after errors if arg is nil, a backtrace will not automatically be displayed (use (:backtrace) for display explicitly.

`(set_module_description MOD DESC)`
: Set the description for the module named MOD.

`(set_server_safe_functions LIST)`
: Sets restricted list to LIST. When restricted list is non-nil only functions whose names appear in this list may be executed. This is used so that clients in server mode may be restricted to a small number of safe commands. [see Section 28.3 [Server/client API], page 138]

`(setenv VARNAME VALUE)`
: Set the UNIX environment variable VARNAME to VALUE.

`(setup_beep_lex)`
: Lexicon derived from the British English Example Pronunciation dictionary (BEEP) from Tony Robinson ajr@eng.cam.ac.uk. Around 160,000 entries.

`(setup_cmu6_lex)`
: Lexicon derived from the CMU lexicon (cmudict-0.6), around 100,000 entries, in the radio phoneset (sort of darpa-like). Includes letter to sound rule model trained from this data, the format of this lexicon is suitable for the UniSyn metrical phonology modules. That is the entries are not syllabified,

`(setup_cmu_lex)`
: Lexicon derived from the CMU lexicon (cmudict-0.4), around 100,000 entries, in the radio phoneset (sort of darpa-like). Includes letter to sound rule model trained from this data, and uses the lexical stress predictor from OALD.

`(setup_cmumt_lex)`
: Lexicon derived from the CMU lexicon (cmudict-0.4), around 100,000 entries, in the radio phoneset (sort of darpa-like). Includes letter to sound rule model trained from this data, and uses the lexical stress predictor from OALD.

`(setup_cstr_lexicon)`
: Define and setup the CSTR lexicon. The CSTR lexicon consists of about 25,000 entries in the mrpa phone set. A large number of specific local entries are also added to the addenda.

`(setup_moby_lexicon)`
: Define and setup the MOBY lexicon. This is derived from the public domain version of the Moby (TM) Pronunciator II lexicon. It can be converted automatically to British English mrpa phoneset which of course is sub-optimal. It contains around 120,000 entries and has part of speech information for homographs.

`(setup_oald_lexicon)`
: Define and setup the CUVOALD lexicon. This is derived from the Computer Users Version of the Oxford Advanced Learners' Dictionary of Current English. This version includes a trained set of letter to sound rules which have also been used to reduce the actual lexicon size by over half, for those entries that the lts model gets exactly the same.

`(socket_open HOST PORT HOW)`
: Open a file descriptor to the BSD socket on HOST at PORT. HOW may be "r" or "w" for a read only or write only filedescriptor. If HOW is unspecified or NIL, "w" is assumed. If HOW is "rw" then a list of two file descriptors is returned, the first for reading the second for writing. Take care when using the bidiectional socket that deadlock doesn't occur.

`(sort-and-dump-docstrings DOCSTRINGS FILEFP)`
: DOCSTRINGS is an assoc list of name and document string var-docstrings or func-docstrings. This very individual function sorts the list and prints out the documentation strings as texinfo list members to FILEFP.

`(sqrt NUM)`
: Return square root of NUM.

`(string-after ATOM AFTER)`
: Returns an atom whose printname is the substring of ATOM's printname which appears after AFTER. This is a wraparound for the EST_String.after function in C++, and hence has the same conditions for boundary cases.

`(string-append STR1 STR2 ...)`
: Return a string made from the concatenation of the print names of STR1 STR2 ...

`(string-before ATOM BEFORE)`
: Returns an atom whose printname is the substring of ATOM's printname which appears before BEFORE. This is a wraparound for the EST_String.before function in C++, and hence has the same conditions for boundary cases.

`(string-equal ATOM1 ATOM2)`
: Returns t if ATOM's printname is equal to ATOM's print name, otherwise it returns nil.

`(string-length SYMBOL)`
: Return the number of characters in the print name of SYMBOL.

`(string-matches ATOM REGEX)`
: Returns t if ATOM's printname matches the regular expression REGEX, otherwise it returns nil.

`(sub_utt ITEM)`
: Return a new utterance that contains a copy of this item and all its descendents and related descendents.

`(substring STRING START LENGTH)`
: Return a substring of STRING starting at START of length LENGTH.

`(sxhash OBJ N)`
: Return hashing value for OBJ, in range n.

`(syl_yn_question utt syl)`
: Return 1 if this is the last syllable in a yes-no question. Basically if it ends in question mark and doesn't start with a wh-woerd. This isn't right but it depends on how much you want rising intonation.

`(symbol-bound? VARNAME)`
: Return t is VARNAME has a value, nil otherwise.

`(symbol-value SYMBOLNAME)`
: Returns the value of SYMBOLNAME, an error is given SYMBOLNAME is not a bound symbol.

`(symbol? DATA)`
: Returns t if DATA is a symbol, nil otherwise.

`(symbolconc SYMBOL1 SYMBOL2 ...)`
: Form new symbol by concatenation of the print forms of each of SYMBOL1 SYMBOL2 etc.

`(symbolexplode SYMBOL)`
: Returns list of atoms one for each character in the print name of SYMBOL.

`(system COMMAND)`
: Execute COMMAND (a string) with the UNIX shell.

`(targets_to_f0 UTT)`
: Make f0 relation, and place an f0 contour in it, using F0 targets from the Target Relation

`(terpri FILEP)`
: Print newline to FILEP, is FILEP is nil or not specified a newline it is printed to stdout.

`(Text UTT)`
: From string in input form tokenize and create a token stream.

`(the-environment)`
: Returns the current (SIOD) environment.

`(tilt_add_intevent utt syl name)`
: Add a new IntEvent related to syl with name.

`(tilt_assign_parameters utt)`
: Assigned tilt parameters to IntEvents, depending on the value of the Parameter tilt_method uses wagon trees (cart) or linear regression models (lr).

`(tilt_assign_parameters_lr utt)`
: Assing parameters (start_f0, tilt, amplitude, peak_pos and duration) to each IntEvent. Prediction by linear regression models

`(tilt_assign_parameters_wagon utt)`
: Assing parameters (start_f0, tilt, amplitude, peak_pos and duration) to each IntEvent. Uses Wagon trees to predict values

`(tilt_assign_params_lr ie lrmodels)`
: Assign the names parameters to ie using the trees and names in trees.

`(tilt_assign_params_wagon ie trees)`
: Assign the names parameters to ie using the trees and names in trees.

`(tilt_map_f0_range utt)`
: In order fo better trained models to be used for voices which don't have the necessary data to train models from the targets may be mapped to a different pitch range. Note this is not optimal as pitch ranges don't map that easily, but the the results can sometimes be better than using a less sophisticated F0 generation model. The method used is to define the mean and standard deviation of the speaker the model was trained on and the mean and standard deciation of the desired speaker. Mapping is by converting the actual F0 value to zscores (distance from mean in number of stddev) and back into the other domain. The variable int_tilt_params is used to find the values.

`(tilt_validate utt)`
: Checks that the predicted tilt parameter fall with reasonable limits and modify them where possible to be more reasonable.

`(time_to_next_vowel syl)`
: The time from vowel_start to next vowel_start

`(tok_allcaps sc)`
: Returns 1 if sc's name is all capitals, 0 otherwise

`(tok_rex sc)`
: Returns 1 if King like title is within 3 tokens before or 2 after.

`(tok_rex sc)`
: Returns 1 if this is a King-like name.

`(tok_roman_to_numstring ROMAN)`
: Takes a string of roman numerals and converts it to a number and then returns the printed string of that. Only deals with numbers up to 50.

`(tok_section_name sc)`
: Returns 1 if sc's name is in list of things that are section/chapter like.

`(tok_string_as_letters NAME)`
: Return list of letters marked as letter part of speech made by exploding NAME.

`(Token UTT)`
: Build a Word stream from the Token stream, analyzing compound words numbers etc as tokens into words. Respects the Parameter Language to choose the appropriate token to word module.

`(Token_Any UTT)`
: Build a Word stream from the Token stream, in a language independent way, which means that all simple tokens should be in the lexicon, or analysed by letter to sound rules.

`(token_end_punc UTT WORD)`
: If punctuation at end of related Token and if WORD is last word in Token return punc, otherwise 0.

`(Token_English UTT)`
: Build a Word stream from the Token stream, for English (American and British English), analyzing compound words, numbers, etc. as tokens into words.

`(token_money_expand type)`
: Convert shortened form of money identifier to words if of a known type.

`(token_no_starting_quote TOKEN)`
: Check to see if a single quote (or backquote) appears as prepunctuation in this token or any previous one in this utterance. This is used to disambiguate ending single quote as possessive or end quote.

`(Token_POS UTT)`
: Assign feature token_pos to tokens thats match CART trees in the variable token_pos_cart_trees. These are used for gross level pos such as identifying how numbers should be pronunced.

`(tok_pos sc)`
: Returns a general pos for sc's name. numeric All digits number float or comma'd numeric sym Contains at least one non alphanumeric month has month name (or abbrev) day has day name (or abbrev) rootname else downcased alphabetic. Note this can be used to find token_pos but isn't used directly as its not disciminatory enough.

`(Token_Spanish UTT)`
: Build a Word stream from the Token stream, for Castillian Spanish, analyzing compound words, numbers etc as tokens into words.

`(english_token_to_words TOKEN NAME)`
: Returns a list of words for NAME from TOKEN. This allows the user to customize various non-local, multi-word, context dependent translations of tokens into words. If this function is unset only the builtin translation rules are used, if this is set the builtin rules are not used unless explicitly called. [see Section 15.2 [Token to word rules], page 75]

`(Token_Welsh UTT)`
: Build a Word stream from the Token stream, for Welsh, analyzing compound words, numbers etc as tokens into words.

`(zerostart sc)`
: Returns, 1 if first char of sc's name is 0, 0 otherwise.

`(track.copy TRACK)`
: Return a copy of TRACK.

`(track.load FILENAME FILETYPE ISHIFT)`
: Load and return a track from FILENAME. Respect FILETYPE is specified and ISHIFT if specified.

`(track.save TRACK FILENAME FILETYPE)`
: Save TRACK in FILENAME, in formar FILETYPE, est is used if FILETYPE is unspecified or nil.

`(tts FILE MODE)`
: Convert FILE to speech. MODE identifies any special treatment necessary for FILE. This is simply a front end to tts_file but puts the system in async audio mode first. [see Chapter 9 [TTS], page 33]

`(tts_file FILE MODE)`
: Low level access to tts function, you probably want to use the function tts rather than this one. Render data in FILE as speech. Respect MODE. Currently modes are defined through the variable tts_text_modes.

`(tts_file_xml FILE)`
: Low level tts processor for XML files. This assumes that element instructions are set up in the variable xxml_elements.

`(find_text_mode FILE ALIST)`
: Search through ALIST for one that matches FILE. Returns nil if nothing macthes.

`(tts_return_to_client)`
: This function is called by clients who wish to return waveforms of their text samples asynchronously. This replaces utt.play in tts_hooks with utt.send.wave.client.

`(tts_text STRING mode)`
: Apply tts on given string. That is, segment it into utterances and apply tts_hooks to each utterance. This is naively done by saving the string to a file and calling tts_file on that file. This differs from SayText which constructs a single utterance for the whole given text.

`(tts_textall STRING MODE)`
: Apply tts to STRING. This function is specifically designed for use in server mode so a single function call may synthesize the string. This function name maybe added to the server safe functions.

`(typeof OBJ)`
: Returns typeof of given object.

`(UniSyn_Duration utt)`
: predicts Segment durations is some speficied way but holds the result in a way necessary for other Unisyn code.

`(Unisyn_Pauses UTT)`
: Predict pause insertion in a Unisyn utterance structure.

`(unwind_protect NORMALFORM ERRORFORM)`
: If an error is found while evaluating NORMALFORM catch it and evaluate ERRORFORM and continue. If an error occurs while evaluating NORMALFORM all file open evaluating NORMALFORM up to the error while be automatically closed. Note interrupts (ctrl-c) is not caught by this function.

`(upcase SYMBOL)`
: Returns a string with the upcased version of SYMBOL's printname.

`(us_db_params)`
: Return parameters of current UniSyn database.

`(us_db_select NAME)`
: Select named UniSyn database.

`(us_diphone_init DIPHONE_NAME)`
: Initialise UniSyn diphone synthesizer with database DIPHONE_NAME.

`(us_ps_synthesis UTT SIGPR)`
: Synthesize utterance UTT using signal processing technique SIGPR for the UniSyn pitch-synchronous synthesizer.

`(us_f0_to_pitchmarks UTT F0_relation PM_relation END_TIME)`
: From the F0 contour in F0_relation, create a set of pitchmarks in PM_relation. If END_TIME is not nil, Extra pitchmarks will be created at the default interval up to this point

`(us_ps_synthesis UTT SIGPR)`
: Synthesize utterance UTT using signal processing technique SIGPR for the UniSyn pitch-synchronous synthesizer.

`(us_td_synthesis UTT FILTER_METHOD OLA_METHOD)`
: Synthesize utterance UTT using signal processing technique SIGPR for the UniSyn pitch-synchronous synthesizer.

`(warp_utterance UTT (Wavefile Pitchmark_file))`
: Change waveform to match prosodic specification of utterance.

`(us_get_synthesis UTT)`
: Construct a unit stream in UTT comprising suitable diphones. The unit stream produced is suitable for immediate use in us_ps_synthesis.

`(us_insert_initial_pause UTT)`
: Always have an initial silence if the utterance is non-empty. Insert a silence segment after the last segment in WORDITEM in UTT.

`(us)insert_pause UTT WORDITEM)`
: Insert a silence segment after the last segment in WORDITEM in UTT.

`(us_list_dbs)`
: List names of UniSyn databases currently loaded.

`(us_make_group_file FILENAME PARAMS)`
: Make a group file from the currently specified diphone set. PARAMS is an optional assoc list and allows specification of the track_file_format (default est_binary), sig_file_format (default snd) and sig_sample_format (default mulaw). This is recommended for LPC databases. For PSOLA based databases the sig_sample_format should probably be set to short.

`(us_mapping UTT method)`
: Synthesize utterance UTT using signal processing technique SIGPR for the UniSyn pitch-synchronous synthesizer.

`(us_unit_concat UTT)`
: Concat coef and wave information in unit stream into a single Frames structure storing the result in the Frame relation

`(us_init_raw_concat UTT).`

`(utt.copy_relation UTT FROM TO)`
: copy relation "from" to a new relation "to". Note that items are NOT copied, simply linked into the new relation

`(utt.copy_relation_and_items UTT FROM TO)`
: copy relation and contents of items "from" to a new relation "to"

`(utt.evaluate UTT)`
: evaluate all the features in UTT, replacing feature functions with their evaluation.

`(utt.evaluate.relation UTT)`
: evaluate all the features in RELATION in UTT, replacing feature functions with their evaluation.

`(utt.feat UTT FEATNAME)`
: Return value of feature name in UTT.

`(utt.features UTT RELATIONNAME FUNCLIST)`
: Get vectors of feature values for each item in RELATIONNAME in UTT. [see Section 14.6 [Features], page 71]

`(utt.id UTT id_number)`
: Return the item in UTT whose id matches id_number.

`(utt.import.track UTT FILENAME RELATION FEATURE_NAME)`
: Load track in FILENAME into UTT in R:RELATION.first.FEATURE_NAME. Deletes RELATION if it already exists. (you maybe want to use track.load directly rather than this legacy function.

`(utt.import.wave UTT FILENAME APPEND)`
: Load waveform in FILENAME into UTT in R:Wave.first.wave. If APPEND is specified and non-nil append this to the current waveform.

`(utt.load UTT FILENAME)`
: Loads UTT with the streams and stream items described in FILENAME. The format is Xlabel-like as saved by utt.save. If UTT is nil a new utterance is created, loaded and returned. If FILENAME is "-" the data is read from stdin.

`(utt.play UTT)`
: Play waveform in utt by current audio method.

`(utt.relation UTT RELATIONNAME)`
: Return root item of relation RELATIONNAME in UTT.

`(utt.relation.append UTT RELATIONNAME ITEM)`
: Append ITEM to top of RELATIONNAM in UTT. ITEM may be a LISP description of an item or an item itself.

`(utt.relation.create UTT RELATIONNAME)`
: Create new relation called RELATIONNAME in UTT.

`(utt.relation.delete UTT RELATIONNAME)`
: Delete relation from utt, it the stream items are not linked elsewhere in the utterance they will be deleted too.

`(utt.relation.feat UTT RELNAME FEATNAME)`
: Return value of FEATNAME on relation RELNAME in UTT.

`(utt.relation.first UTT RELATIONNAME)`
: Returns a the first item in this relation.

`(utt.relation.items UTT RELATIONNAME)`
: Return a list of stream items in RELATIONNAME in UTT. If this relation is a tree, the parent streamitem is listed before its daughters.

`(utt.relation.last UTT RELATIONNAME)`
: Returns a the last item in this relation.

`(utt.relation.leafs UTT RELATIONNAME)`
: Returns a list of all the leafs in this relation.

`(utt.relation.load UTT RELATIONNAME FILENAME)`
: Loads (and creates) RELATIONNAME from FILENAME into UTT. FILENAME should contain simple Xlabel format information. The label part may contain the label proper followed by semi-colon separated pairs of feature and value.

`(utt.relation.present UTT RELATIONNAME)`
: Returns t if UTT caontains a relation called RELATIONNAME, nil otherwise.

`(utt.relation.print UTT NAME)`
: print contents of relation NAME

`(utt.relation.remove_feat UTT RELNAME FEATNAME)`
: Remove FEATNAME on relation RELNAME in UTT.

`(utt.relation.remove_item_feat UTT RELNAME FEATNAME)`
: Remove FEATNAME on every item in relation RELNAME in UTT.

`(utt.relation.set_feat UTT RELNAME FEATNAME VALUE)`
: Set FEATNAME to VALUE on relation RELNAME in UTT.

`(utt.relation_tree UTT RELATIONNAME)`
: Return a tree of stream items in RELATIONNAME in UTT. This will give a simple list if the relation has no ups and downs. [see Section 14.5 [Accessing an utterance], page 68]

`(utt.relationnames UTT)`
: List of all relations in this utterance.

`(utt.resynth LABFILE F0FILE)`
: Resynthesize an utterance from a label file and F0 file (in any format supported by the Speech Tool Library). This loads, synthesizes and plays the utterance.

`(utt.save UTT FILENAME TYPE)`
: Save UTT in FILENAME in an Xlabel-like format. If FILENAME is "-" then print output to stdout. TYPE may be nil or est_ascii

`(utt.save.f0 UTT FILENAME)`
: Save F0 of UTT as esps track file in FILENAME.

`(utt.save UTT RELATIONNAME FILENAME EVALUATE_FEATURES)`
: Save relation RELATIONNAME in FILENAME in an Xlabel-like format. If FILENAME is "-" then print output to stdout.

`(utt.save.segs UTT FILE)`
: Save segments of UTT in a FILE in xlabel format.

`(utt.save.til_events UTT FILENAME)`
: Save tilt events in UTT to FILENAME in a format suitable for ev_synth.

`(utt.save.track utt filename relation feature)`
: DEPRICATED use trace.save instead.

`(utt.save.wave UTT FILENAME FILETYPE)`
: Save waveform in UTT in FILENAME with FILETYPE (if specified) or using global parameter Wavefiletype.

`(utt.save.words UTT FILE)`
: Save words of UTT in a FILE in xlabel format.

`(utt.send.wave.client UTT)`
: Sends wave in UTT to client. If not in server mode gives an error Note the client must be expecting to receive the waveform.

`(utt.set_feat UTT FEATNAME VALUE)`
: Set feature FEATNAME with VALUE in UTT.

`(utt.synth UTT)`
: The main synthesis function. Given UTT it will apply the functions specified for UTT's type, as defined with deffUttType and then those demanded by the voice. After modules have been applied synth_hooks are applied to allow extra manipulation. [see Section 14.2 [Utterance types], page 65]

`(utt.type UTT)`
: Returns the type of UTT.

`(utt.wave UTT)`
: Get waveform from wave (R:Wave.first.wave).

`(utt.wave.resample UTT RATE)`
: Resample waveform in UTT to RATE (if it is already at that rate it remains unchanged).

`(utt.wave.rescale UTT FACTOR NORMALIZE)`
: Modify the gain of the waveform in UTT by GAIN. If NORMALIZE is specified and non-nil the waveform is maximized first.

`(Utterance TYPE DATA)`
: Build an utterance of type TYPE from DATA. Different TYPEs require different types of data. New types may be defined by defUttType. [see Section 14.2 [Utterance types], page 65]

`(voice-location NAME DIR DOCSTRING)`
: Record the location of a voice. Called for each voice found on voice-path. Can be called in site-init or .festivalrc for additional voices which exist elsewhere.

`(voice.describe NAME)`
: Describe voice NAME by saying its description. Unfortunately although it would be nice to say that voice's description in the voice itself its not going to work cross language. So this just uses the current voice. So here we assume voices describe themselves in English which is pretty anglo-centric, shitsurei shimasu.

`(voice.description NAME)`
: Output description of named voice. If the named voice is not yet loaded it is loaded.

`(voice.list)`
: List of all (potential) voices in the system. This checks the voice-location list of potential voices found be scanning the voice-path at start up time. These names can be used as arguments to voice.description and voice.describe.

`(voice.select NAME)`
: Call function to set up voice NAME. This is normally done by prepending voice_ to NAME and call it as a function.

`(voice_rab_diphone)`
: Set up the current voice to be a British male RP (Roger) speaker using the rab diphone set.

`(voice_reset)`
: This resets all variables back to acceptable values that may affect voice generation. This function should always be called at the start of any function defining a voice. In addition to reseting standard variables the function current_voice_reset will be called. This should always be set by the voice definition

function (even if it does nothing). This allows voice specific changes to be reset when a new voice is selection. Unfortunately I can't force this to be used.

`(wagon ITEM TREE)`
: Apply the CART tree TREE to ITEM. This returns the full predicted form, you need to extract the value from the returned form itself. [see Section 25.2 [CART trees], page 120]

`(wagon_predict ITEM TREE)`
: Predict with given ITEM and CART tree and return the prediction (the last item) rather than whole probability distribution.

`(wave.copy WAVE1 WAVE2)`
: Destuctively append WAVE2 to WAVE1 and return WAVE1.

`(wave.copy WAVE)`
: Return a copy of WAVE.

`(wave.info WAVE)`
: Returns assoc list of info about this wave.

`(wave.load FILENAME FILETYPE SAMPLETYPE SAMPLERATE)`
: Load and return a wave from FILENAME. Respect FILETYPE is specified if not specified respect whatever header is on the file. SAMPLETYPE and SAMPLERATE are only used if FILETYPE is raw.

`(wave.play WAVE)`
: Play wave of selected audio

`(wave.resample WAVE NEWRATE)`
: Resamples WAVE to NEWRATE.

`(wave.rescale WAVE GAIN NORMALIZE)`
: If NORMALIZE is specified and non-nil, maximizes the waveform first before applying the gain.

`(wave.save WAVE FILENAME FILETYPE SAMPLETYPE)`
: Save WAVE in FILENAME, respecting FILETYPE and SAMPLETYPE if specifed if these last two arguments are unspecified the global parameters Wavefiletype and Wavesampletype are used. Returns t is successful and throws an error if not.

`(Wave_Synth UTT)`
: Generate waveform from information in UTT, at least a Segment stream must exist. The actual form of synthesis used depends on the Parameter Synth_Method. If it is a function that is applied. If it is atom it should be a SynthType as defined by defSynthType [see Section 14.2 [Utterance types], page 65]

`(wfst.load NAME FILENAME)`
: Load a WFST from FILENAME and store it named NAME for later access.

`(wfst.trasduce WFSTNAME INPUT)`
: Transduce list INPUT (or exploded INPUT if its an atom) to a list of outputs. The atom FAILED is return if the transduction fails.

`(while COND . BODY)`
: While COND evaluates to non-nil evaluate BODY.

`(Word utt)`
: Construct (synthesis specific) syllable/segments from Word relation using current lexicon and specific module.

`(xml_register_id PATTERN RESULT)`
: Add a rule for where to find XML entities such as DTDs. The pattern is a regular expression, the result is a string with substitutions. If the PATTERN matches the a PUBLIC or SYSTEM identifier of an XML entity, the RESULT is expanded and then used as a filename.

`(xml_registered_ids)`
: Return the current list of places to look for XML entities.

`(xxml_attval ATTNAME ATTLIST)`
: Returns attribute value of ATTNAME in ATTLIST or nil if it doesn't exists.

`(xxml_synth UTT)`
: This applies the xxml_hooks (mode specific) and tts_hooks to the given utterance. This function should be called from xxml element definitions that signal an utterance boundary.

Index

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

Table of Contents