Sound Packs

A Proposal and Offer to the Asterisk Community

gm voices®
your voice to the world

by Steve Murphy and GM Voices

History

About 3 years ago, GM Voices contacted me to help them decide what would need to be done in order for them to provide high-end sound sets for the Asterisk market. Looking at the current state of Asterisk, and what would be necessary to allow them to play sounds the way they would like, I determined that it would not be possible unless Asterisk had considerable changes. The name for this project was the "Concatenation Engine". I then proposed the following setup, but they could not find a way to fund the project.

I then actually coded (in Java), a major portion of the software involved, in order to provide "proof of concept", but never really totally finished it, as I had no assurance that I was not wasting my time. A really good solution just cries for attention, though, and I asked that the results and work done so far be released to the Asterisk community. If they think it a good idea also, then they might be willing to help complete and implement it. I have just recently received permission from GM Voices to do just that.

Acknowledgements

Hardly any of the ideas presented in this document are new and groundbreaking! After my initial definition of SayScripts was completed, and the document produced, I did deeper research into Asterisk to estimate how much work it would be to convert the existing code to a SayScript base. It was then that I discovered the code Luigi Rizzo had committed to 1.4 in 2006, as part of the Playback app. It is totally undocumented, and virtually hidden in the code, but the tables outlined in the say.conf file are very, very close to what I had outlined for SayScripts. I don't want to seem self-aggrandizing, but perhaps great minds think alike?

And, I must also acknowledge the work of Tilghman Lesher, in the formatted date code of Asterisk, which also allows files to be intermingled with the letters that call out functions to pronounce things like the day of the week, and so forth, which was very close to what I designed for the say_sentence format string.

And, of course, I must acknowledge the work of the authors of the GNU gettext package, a work of art for the internationalization (i18n), and localization (l10n) of all the programs in the GNU textual domain. They can be credited with the %-constructs and argument manipulation in printf-like format strings, and the whole translation workflow scheme.

The Problems:

Adding Languages and Code

Asterisk has a problem with languages. As far as I can tell, only one language, vi, has been introduced in the last 3 years; some changes to others have been made, etc. …

To fully realize what is involved with adding a new language, pardon me while I describe the current situation in Asterisk, in how it "plays" sound files to users, and how the localizations are arranged...

Existing Asterisk Functions That Play Speech Files

ast_streamfile(chan, filenamestr, lang)

initiates playing the given file, passes in the language string, to find the proper files to play. Returns -1 if the file can't be opened/found.

ast_waitstream(chan, interruptstring)

will wait for the sound file to finish playing. If any of the keys in interruptstring are returned, the key pressed will be returned. A following call to stopstream should cut off the playing.

ast_stopstream(chan)

Stops and closes the currently playing stream file.

ast_play_and_wait(chan, filenamestr)

Calls the above 3 functions to play the file and wait for either the file finish playing, or or return the interrupt character, if a matching interrupt key was pressed. In this case, ast_play_and_wait passes "AST_DIGIT_ANY" to ast_waitstream().
Returns -1 if the file can't be opened/found.

ast_app_getdata(chan, prompt, stringbuffer, stringbuffersize, timeout)

Uses streamfile to pronounce the prompt; then waits for timeout seconds to read the input and place all pressed keys into the stringbuffer. Used about 16 times in various Asterisk locations; Uses ast_readstring(c, s, len, timeout, ftimeout, enders).

The above routines are the main mechanism within asterisk for interaction with users. They are used in the following source files with roughly the mentioned frequency:

apps/app_voicemail.c: ~404 (calls say_number 17 times) (say_digit_str 5 times), say_date_with_format 12 times) (say_counted_noun) (say_counted_adjective) (say_character_str) (ast_app_getdata)
/main/say.c: 122
apps/app_meetme.c: 53 (calls ast_say_number 6 times) (say_digits 2 times) (app_getdata 3 times)
apps/app_minivm.c: 20 (calls ast_say_digitstr)
apps/app_dial.c: 9
apps/app_followme.c: 8
apps/app_directory.c: 6 (calls ast_say_character_str 3 times)
main/app.c: 10
apps/app_privacy.c: 5
apps/app_dial.c: 4
main/features.c: 4
channels/chan_agent.c: 4 (calls app_getdata 3 times)
apps/app_playback.c: 3 (defines/calls all the play_ functions)
addons/app_saycountpl.c: 3
apps/app_talkdetect.c: 3
apps/app_confbridge.c: 2 (calls say_number 8 times) (calls app_getdata)
apps/app_record.c: 2
apps/app_authenticate.c: 2 (calls ast_app_getdata)
apps/app_queue.c: 2 (calls say_number 6 times)
apps/app_chanspy.c: 2 (calls say_character_str, and say_digits)

apps/app_pakandannounce.c: 2 (say_digits)
res/res_agi.c: 2 (calls say_number once, as welll as all the other say_ funcs) (calls app_getdata)
apps/app_readexten.c: 1
apps/app_read.c: (calls app_getdata)
apps/app_page.c: 1
apps/app_dictate.c: 1 (calls say_number once) (ast_app_getdata)
main/file.c: 1 (defined here)
channels/chan_sip.c: 1
channels/sig_analog.c (say_digit_str)q
channels/chan_dahdi.c1Gq
apps/app_skel.c :1 (this is a call to ast_say_number())
main/channel.c (calls ast_say_number 3 times)
main/pbx.c (calls say_number once) (also, say_digit_str, say_character_str, and say_phonetic_str)
apps/app_saycounted.c: (say_counted_noun) (say_counted_adjective)
apps/app_sayunixtime.c: (say_date_with_format)
channels/chan_mgcp.c: (say_digit_str)
apps/app_test.c: (calls app_getdata 3 times)
apps/app_dahdibarge.c: (app_getdata)

Existing Asterisk "say" functions

say_character_str_full (chan, str, ints, lang, audiofd, ctrlfd)

pronounces each char in the string in sequence, *,#,!, @, $, -, ., =, +, /, space, 0-9, A-Z
inter-word spacing provided by each soundfile. uses the sound files in digits/ and letters/
– lang is used to select among different sets in different subdirs in /var/lib/asterisk/sounds
– Implementations in en, cs, da, de, en_GB, es, fr, he, it, nl, no, pl, pt, se, zh, gr, ru, ka, hu, th, ur, vi.

say_phonetic_str_full (chan, str, ints, lang, audiofd, ctrlfd)

pronounces each char in the string in sequence, *,#,!, @, $, -, ., =, +, /, space, 0-9, A-Z
inter-word spacing provided by each soundfile. uses the sound files in digits (0-8), letters (the graphic chars), and phonetic/ for 9, and all the alphabetic letters
– this func plays the "Alpha-bravo-charley- niner" stuff for each letter in the alphabet, as in the military
– lang is used to select among different sets in different subdirs in /var/lib/asterisk/sounds

say_digit_str_full (chan, str, ints, lang, audiofd, ctrlfd)

pronounces each char in the string in sequence, *,#, -, 0-9
nter-word spacing provided by each soundfile. uses the sound files in digits/.
– lang is used to select among different sets in different subdirs in /var/lib/asterisk/sounds

say_number_full_* (chan, num (int), ints, lang, audiofd, ctrlfd)

– implementations for en, cs, da, de, en_GB, es, fr, he, it, nl, no, pl, pt, se, zh, gr, ru, ka, hu, th, ur (english rules apply by default for those not explicitly coded, but use their own language sound files)
– Turns 1203 into "1 thousand 2 hundred and 3"
– handles millions, but no higher.

say_enumeration_full_* (chan, num (int), ints, lang, audiofd, ctrlfd)

– this routine plays "1st" "2nd", "3rd"... "14th" and so on. "six-hundred,sixty,4th", etc.
– implementations for en, da, de, he, vi (english rules apply by default for those not explicitly coded, but use their own language sound files)
– handles billions

say_date_* (chan, time_t, ints, lang)
– implementations for en, da, de, fr, he, nl, pt, gr, ka, hu, th (english rules apply by default for those not explicitly coded, but use their own language sound files)

say_date_with_format_* (chan, time_t, ints, lang)
– implementations for en, da, de, es, fr, he, it, nl, pl, pt, zh, gr, th, vi; english rules apply by default for those not explicitly coded, but use their own language sound files)

The codes are:

* A, a – pronounce 'sunday' thru 'saturday', according to timeval passed
* B, b, h – pronounce 'january' thru 'december'.
* d, e – say the day of the month, from "first" to "thirtyfirst"
* Y – say the year, from "nineteen oh one" thru "two thousand ... whatever"
* I l (Cap i, little L) – say the hour (12-hour format) from "oh" to 12.
* H k – say the hour in 24-hour format. H as oh-8, k says just '8'. 20+ is "twenty" plus "whatever remains"
* m – say the month number in "nth" format, as in "1st", "2nd", "3rd",..., "twelfth"
* M, N – say the minutes; if 0, M will say "oclock", but N will say "hundred". An "oh" precedes mins less than 10.
* P, p, – say "AM", or "PM"
* Q – relative time; It says "today", or "yesterday", within the last week, it says the weekday name; within a month, it says "Sunday, October 3rd"'; under 6 months, it leaves out the day of the week, and over six month, it says the month, day, and year. (at least, in US english).
* q – seems to do the exact same as %Q.
* R – equiv of HM, "oh" "eight" "thirty" "three"
* S – seconds, "zero", "oh" "five", and so on.
* T – equiv of HMS
* Filenames in single quotes are played in place.

say_time_* (chan, time_t, ints, lang)
– implementations for en, de, fr, he, nl, pt, pt_BR, zh, gr, ka, hu, th (english rules apply by default for those not explicitly coded, but use their own language sound files) "ten" "oh" "five" "am"

say_datetime_* (chan, time_t, ints, lang)
– implementations for en, de, fr, he, nl, pt, pt_BR, zh, gr, ka, hu, th (english rules apply by default for those not explicitly coded, but use their own language sound files)

say_datetime_from_now_* (chan, time_t, ints, lang)
– implementations for en, fr,pt, ka, he (english rules apply by default for those not explicitly coded, but use their own language sound files). Past times only. Says "today", "yesterday", and more explicit dates the further in the past they get. All future times are pronounced the "long" way.

These 3 functions below are the main functions used throughout Asterisk to play soundfiles: They are also used by the "say" functions to play files:

```
int ast_streamfile(chan, filename, language)  /* initiate playing a stream-file. */
int ast_waitstream(chan, interrupt_chars);     /* wait until file is finished, or an interrupt key hit */
int ast_stopstream(chan)  /* stop, close the stream */
```

These functions are based on the steamfile() functions above:

int play_and_wait() plays a file with AST_DIGIT_ANY.
int stream_and_wait()
int wait_file2()
say_and_wait() calls the ast_say_number().)

The above say_ functions are implemented for the following languages:

Lang (3)	language	Rank/num	number	enum	date	Date_form	time	datetime	Datetime_ now	vmintro
engv	en	3/328	X	X	X	X	X	X	X	X
ces	cs (Czech)	81/9.5	X							X
dan	da (danish)	118/5.6	X	X	X	X				
deu	de(german)	10/90.3	X	X	X	X	X	X		X
	en_GB		X							
spa	es (span)	2/329	X			X				X
fra	fr (French)	16/67.8	X		X	X	X	X	X	X
ell	gr (greek?)	68/13.1	X			X	X	X		X
heb	he(Hebrew)	121/5.3	X	X	X	X	X	X	X	X
hun	hu (hunga)	73/12.5	X		X		X	X		
kat	ka (georg)	138/4.3	X		X		X	X	X	
ita	it (italian)	19/61.7	X			X				X
nld	nl (dutch)	42/21.7	X		X	X	X	X		X
nor	no (norwg)	132/4.6	X							X
pol	pl (polish)	23/40.0	X			X				X
por	pt (portu)	7/178	X		X	X	X	X	X	X
	pt_BR		X				X	X		X
rus	ru(Russian)	8/144	X							X
swe	Se (Swedish is really sv!)	88/8.3	X							X
nod	th (thai)	110/6.0	X		X	X	X	X		
urd	ur (urdu)	20/60.6	X							
zho	zh (chin)	1/845	X			X	X	X		
Ukr	Ua (Ukrain)									X
	Total	/2234.8								

In the chart above, the Rank is the placing based on the number of native speakers. The num is the number of native speakers, in millions. It is unclear if the non-checked routines are that way because the defaults are sufficient, or they were simply not yet written. Language codes are from ISO 639-1, and ISO-639-2 if no 2-letter code is available, and country codes are from ISO 3166.

Another place where localizations are coded: **vm_intro(channel*, ast_vm_user *, vm_state *)**

This function has localizations to cs, de, en, es, he, zh, it, fr, nl, pt, pt_BR, gr, pl, se(should be sv!), no, ru?, and ua? (ukranian); also

There is also a newer func, **vm_intro_multilang()**, with a gender arg, that hopes to replace all the above localizations. Quite a bit of thought went into the multilang() version, but it still looks like, as more languages are introduced, that it or other funcs will have to be updated to take care of what could be unexpected requirements of each new language.

According to Ethnologue, the say_ language set covers 37.5% of Earth's population, and 0.29% of the 6909 living languages on Earth.

There are 389 languages spoken by 1 million or more speakers. If Asterisk implemented all these languages, it would provide utterances understood by 94% of all the people on Earth.

Yet to be provided will be Arabic (221 million speakers), Hindi (182 mil), Bengali (181 mil), Japanese (122 mil), Javanese (84 mil), Lahnda [Punjabi, Saraiki] (78.3 mil), Telugu (69.8 mil), Marathi (68.1 mil), Korean (66.3 mil), and so on.

Say.c has grown to over 8000 lines of code, but the number of languages introduced to Asterisk has not grown significantly in the last 3 years. I might propose that one of the main reasons so few languages have been introduced is the relatively high cost of introducing a new language:

1. The cost of recording studio-quality versions of each of the 450+ sound files in the core set.

2. The cost of translation from the current English version.

3. The cost of writing language specific versions of the say- functions that use those files.

4. The cost of working around the complexities introduced by the fixed-order sequences that are embedded in the current code.

File substitution vs. File Order

To introduce a new language, the standard approach has been simply to override the English versions of the sound files with equivalent verbiage in the new language. But, because different languages use different word order, this results in some rather nasty problems. I call this file-substitution method a translation "by phrase".

Minimal Sound Set Algorithms for Numbers, Sequences.

Right now, ALL languages use the same fundamental algorithms to generate speech for things like numbers, which basically use what I call a "minimal sound set". Speech is broken up into distinct words, and each is recorded separately in its own sound set. The number 1, for example, is pronounced in exactly one sound file (number/1). Words like "Ten", "Thirteen", "hundred", "thousand", etc are all recorded in exactly one file each, and the algorithms pick and choose the files in an appropriate order to generate utterances like "three hundred forty seven". This is well and good, but as I have learned, minimal sound file sets don't always yield the most natural sounding speech.

The Solution: Sound Packs

So now, let's discuss what I call the "solution" to these problems. Something that will make translation and implementation of new languages and localizations less costly and easier to do, something that will reduce the cost of introducing new languages and speech technologies. We would like to introduce the concept of a Sound Pack in the following pages.

What is a Sound Pack? It is merely a tar file that contains a set of files that fully define a new language to Asterisk. It is meant to to be unloaded (installed?) into /var/lib/asterisk.

A new Paradigm: Unit of Sound (and translation): The Sentence and NOT the File (or Phrase).

One source of trouble has been attempting to translate phrases, which can be shared between sentences by playing the same file. It is far easier to translate sentences, which provide all the pieces necessary to determine factors like gender, context, and so forth.

New SaySentence() core function

int ast_say_sentence(struct channel *chan, char *ecodes, char *format, ...)

Where ecodes is a string of characters that will interrupt the output sequence. If one of these characters are entered by the listener during the playback, the playback is immediately terminated, and the pressed key value is returned from the function. If the string is empty, or a null pointer, then the playback cannot be interrupted, and 0 will be returned when the sentence is finished playing.

Example:

ret = ast_say_sentence(chan, "#*", "<you_transferred> %m <to_your_account> %[o]d", balance, transaction_date);

where the first arg is a string of characters that will 'break' the playback. If the listener hits * or # during the playing of the sound files, the playback is immediately terminated, and that char (* or #) is returned from say_sentence.

A -1 returned as a result indicates an error occurred.

where anything in <> is a phrase file to play; %m runs the say_money script, which should take the balance argument, and interpret it into a string of sounds that will include the "dollars, and ... cents" stuff. %d will play the date specified by transaction_date, [o] saying it should play "on Wednesday, july 24th, 2009". **In general, options to any % construct are placed within square brackets.**

The following format specifiers will be provided by say_sentence:
%**m** -- the SayScript [money] will be run. The argument should be in the format of ddd.cc,
Where ddd is any number of dollar digits, followed by a '.', and two 'cents' digits.

%[o|f|a]**d** – SayScript [date] will be run, and provided with the corresponding date argument, and the o, or f, or a option, to choose "on", "for", or "as of" respectively, as a leading word for the date. If none are chosen, the default of "on" will be used.

In the current version, the date will most likely not include this leading word. Corresponds to the say_date() functions.

%**t** – SayScript [time] will be run, and provided with the timeval you supply in the say_sentence args for this variable. Corresponds to the say_time() functions.

%**n** – SayScript [number] will be run, and provided with the number you supply. Corresponds to the say_number() and say_number_full routines.

%**i** – SayScript [digit_str] will be run, supplied with the string you provide. Non-digits in the string may be ignored. Corresponds to the say_digit_str() functions.

%**s** – SayScript [alphanum_str] will be run, supplied with the string you provide. Non alpha/Non-digit characters may be ignored. Corresponds to the NEW say_alphanum_str() function(s).

%**c** – SayScript [char_str] will be run, supplied with the string you provide. Corresponds to the say_character_str() functions.

%**p** – SayScript [phonetic_str] will be run. Corresponds to the say_phonetic_str() functions.

%**e** – SayScript [enumeration] will be run. Corresponds to the say_enumeration() functions. (Should we have called this ordinal?)

%**D** – SayScript [date_time] will be run, with the timeval you supply. Corresponds to say_datetime() set of routines.

%[date format string]**f** – Date format. The Date format string corresponds exactly to what you would supply to say_date_with_format() (see page 5). The new say_date_with_format will play any sound files specified, and call the SayScript sections that correspond to the format letters: date_format_A, date_format_S, and so forth. A corresponding timeval arg should be supplied. Corresponds to the say_date_with_format() functions.

%r -- SayScript [rel_date_time] will be run. Corresponds to the say_datetime_from_now routine.

%z -- the corresponding argument is expected to be a file path to play. This is for the rare instances where a different file name may need to be played, depending on the circumstances. Example: say_sentence("%z <costs> %m.", get_correct_file_name(ind), "129.95"); The return value of get_correct_file_name(ind) might be "the_bus", or "the_car", for example. Thus, either "the_bus.gsm", or "the_car.gsm" will be played. Never use %z where a constant is passed as an argument. In such a case, use the <filename> construct instead. I cannot stress enough that this construct should not be used. Issue two (or more) different SaySentence requests rather than use this construct, as it will hide the utterances, and make it hard to translate. Remember this: The purpose of SaySentence() is to facilitate sound file mapping and translation, not to save coders some time or make the code neater. (Although, I would think, it will actually do this!)

I purposefully leave the file names directly in the format string where possible (rather than use %z), because we will need to have this format string uniquely identify this utterance.

Say_sentence will automatically add these options to the SayScript calls made within it:

- B – If the first thing in the format is a % construct, that SayScript call will have a B added to its options, to indicate that it is occurring at the beginning of a sentence. If a '.' (period) occurs in the sentence, and is followed by a % construct, B will be added to the options also.
- M – This option is added to all % constructs not at the beginning or end of a sentence.
- E – if a %-construct occurs at the end of a sentence (immediately preceding a period, or the last item in the format specifier, then say_sentence will add an E to the options for that SayScript invocation.

The SayScript is free to use or ignore these options.

New Dialplan Application SaySentence

SaySentence(interrupt_chars,"SaySentence Format String", <arguments, if any>)

where interrupt_chars is a set of keypad keys that are allowed to interrupt the announcement.

The "SaySentence Format String" is enclosed in quotes and follows the same rules and features as described for the internal function (ast_say_sentence).

Arguments are the values to associate with the %- constructs in the generated sounds.

Introducing the Sound Pack

– The Sound Pack Name

The method Asterisk currently uses to hierarchically group locales by name is useful, and there is no reason to change it, in our opinion. The algorithm to try matching directory names first by a complete match, and then by sequentially removing the last '_' character in the locale name, and everything after that, and attempting to match that, and so forth, is honored in the SaySentence() code. The writer of a sound pack should generate a locale name for the sound pack that will uniquely identify the sound pack's files. We suggest using the ISO approved language names, along with company, speaker name, etc, so that multiple products from the same organization can be differentiated, and so that SayScripts, scripts, and translations do not have to be repeated, and can be shared.

– The Sound Pack Contents

1. A set of Sound Files (in ./sounds/<locale-name>/)
2. A file to define the "script" of what is said in each file (in ./sounds/<locale-name>/script.txt)
3. Translation File(s) (in ./translation/<locale-name>.translationset
4. SayScript(s) (in ./language/<locale-name>.sayscript.

More about the above files will now be discussed.

Details of the script.txt file

This file follows a simple format. Comments begin with a ';'. Blank lines are ignored. Every file in a sound set should be included, one file per line. The file name or path relative to its locale directory is first, then a colon, then any number of spaces, and a script for what is said in the file. Right now, no special syntax is provided for annotations, and so forth. Perhaps, in the future, such could be standardized. This pretty much duplicates the current format of the .txt files included with Asterisk default core and extra sound sets.

Details of Translation file

As previously mentioned, the translation files provide a mapping from the language of the dialplan or source code (not necessarily English!), to another language or locale. If all that is needed is to override the sound files, then no mappings are necessary, and the normal file override mechanism will handle the situation. But, with more complex sound sets, this becomes less an option. In this section, I will elaborate on the format of the translation file, and the different mapping features that can be performed.

As an example, let's consider this say_sentence() call:

say_sentence("<you_transferred> %m <to_your_account> %[o]d.", ${balance}, ${transaction_date});

A scanner can be written to scan the Asterisk source code and all the dialplan (traversing all included files), that will find each call to say_sentence() and extract the format strings into a new translation file.
In this case, let's consider German. Such a scanner would make this entry in the file translation/de_DE:

[**you_transferred> %m <to_your_account> %[o]d.]**

Just below it, you would be expected to supply the translation string. If the translation string uses the same elements in the same order, then you might consider providing direct translation override sound files in /var/lib/asterisk/sounds/de_DE, that have the same name as the sound files in /var/lib/asterisk/sounds/, or /var/lib/asterisk/sounds/en, but contain utterances in German instead of English, and you don't need to supply a translation in /var/lib/asterisk/translate/de_DE at all. If NO translation required more than a simple sound file override, then you won't even need to define a /var/lib/asterisk/translate/de_DE file at all!

But things aren't always so simple. German tends to "kick" verbs to the end of the sentence, and so, the translator may want to re-order the parts of the sentence and form this kind of translation for this sentence:

[**you_transferred> %m <to_your_account> %[o]d.]**
"%[o]d, <into_your_account>, %m <was_transfered_by_you>"

where "in_your_account.wav" and "was_transferred_by_you.wav" are new files in your target language locale dir (/var/lib/asterisk/sounds/en_DE) that will give the desired natural affect.

Now, you will note in the above that %d and %m have switched order.

The software should be able to handle juggling the arguments as long as it can find a direct match. Thus, since only one %m and one %d is given, and they match args, then they can be successfully matched. But, if you have more than one arg of the same type, then you can specify a numeric index to disambiguate: assume that implicitly, every % specifier in the English format string is issued a sequential index starting at 1, from left to right, in the original say_sentence string. The translation string can then be written to say:

[**you_transferred> %m <to_your_account> %[o]d.]**
"%2[o]d, <in_your_account>, %1m <was_transfered_by_you>."

We've noted that often, the translation can vary depending on the value of the variables, particularly numbers.

Consider this usage of say_sentence():

say_sentence("<you_put> %n <boxes_into_the_truck>.", ${num_boxes});

which might be so translated:

[<**you_put> %n <boxes_into_the_truck>.**]
{0} "<you_didnt_put_any_boxes_in_the_truck>."
{1} "<you_put_a_box_in_the_truck>".
{2} "%n <boxes_in_the_truck_by_you_were_put>."

If more than one number is included in the sentence, and each would end up requiring such a permutation, then it is advised that you split up the sentence into two sentences, each containing a single number, so each can have its own set of permutations.

This is probably a poor example, because perhaps the English version could be split up into separate sentences to provide the same nice-sounding utterances, and each sentence would have a more direct
Translation to the target language. But, even so, the way a sentence is translated could easily depend on the values used.

Details of SayScripts

SayScript is an interpreted language. It closely resembles the tabular formats already developed for this purpose, but adds features that cause it to transcend from a table of data to an algorithm description.

Ugh! You might say. "Not another language!". We feel your pain. "Why not just use an already developed full-blown language like perl, PHP, Ruby, shell/bash, lua, lisp? Shoot, even an AEL interpreter might suffice!". We might have easlily taken this approach, but consider these facts:

Firstly, most translators aren't going to make good programmers. Another example of this would be web-site graphics designers and web-site programmers. Yes, there exist a few folks that can do both, but they are the exception, not the rule.

The biggest hurdle here will be the programming language. The simpler that language, the lower the hurdle. Let's face it. The job of turning a number into a list of files to play need not be too complex. But learning even the basics of Perl, for example (enough to read code), is a pretty big task!

So SayScript has very simple variables, operates on very restricted data model, rejects recursion (which, by the way, it appears a fair number of programmers have trouble with!), will allow argument passing, uses simple pattern matching and conditionals to achieve the same results as the C-coded routines in Asterisk (which are recursive). We hope that even non-programmer translators with the help of examples, can successfully develop SayScripts for their languages or methodologies.

At the lowest level, all the say_* functions in Asterisk, and all the %*x* primitives in say_sentence will rely on SayScripts to get the job done for the language at hand. Rather than hard-coding the say_ routines in C, and running them in Asterisk, SayScript will allow translators to write pronunciation rules for numeric strings, char strings, money amounts, numbers, dates, etc, for their language, without compiling anything into Asterisk, and will let them test and verify their scripts with standalone programs, without having to run Asterisk.

Since we cannot rewrite all the current say_ functions defined for languages, we will keep the functions in the Asterisk source until the SayScripts for that language are submitted. Thus, the say_ functions for non-English languages will be replaced over time, as the community finds the time.

SayScripts will be written to handle both GM Voices methodologies, as well as current Asterisk usage for English, as a starting point.

SayScript Specification and Examples

A SayScript file is intended to allow multi-lingual specifications of how to algorithmically "say" numbers and strings in different languages, even if the methodologies are quite different, from building blocks of individual words and phrases.

General specification syntax

A SayScript file is composed of comments, and one or more SayScripts.

A Sayscript is composed of a header and body.

The header is a square-bracketed set of comma-separated names. If more than one name is present, they are aliases of each other. These names must all be unique to the SayScript file. The header must not contain any newlines-- it must occupy a single line. No spaces are allowed within the brackets.

The remainder of the SayScript is the body, and is composed of statements, one per line. Blank lines are OK, and Comments (preceded by a semicolon), are allowed before and after statements, and before and after headers.

The system requires that certain SayScript names be defined, as described below. But

other, user-defined names can be defined and called from the required SayScripts.

[identlist] ;; a header telling which utterance is being described.
<identlist> is a single identifier, or a comma separated list of identifiers.
If more than one ident is provided, then they are treated as aliases.
An ident can have one of the following values:
silence ;; a special section-- silence files to use for what notation; no SayScript here.
number
digit_str
alphanum_str
char_str
money
phonetic_str
enumeration
date
time
date_time
date_format_[AaBbhYdeIlhkmMNPpQqRST]
(and combinations of these as appropriate for your vocabulary)
rel_date_time

– other ident names can be declared, and referenced via << >> (see below) notation, but the above are required for a working system. Such other names are not invoked from SaySentence() calls directly, but if the name is composed of '%', and an unused format character, A-C, E-Z or a-b,g-h,j-l, o,q-r,u-y, then you reference them from within SaySentence format strings. These extra, undefined formats can be used to invoke special sayscripts. For instance, a number pronouncing SayScript that gives only 3 or 4 digits of precision... it might say "approximately 1.34 billion" instead of a complete 10-digit number.

– When a SayScript is invoked for a particular header, it can be passed:
1. An option string. These options can be accessed via the expressions outlined below (see {opt:x}),
or, tested by Ops (see OPT and NOTOPT below).

2. A value, consisting of one of the following:
    a. a NUMBER, in the form of a string, and for the [money] section, a floating point number. Since the number is stored both as a string and as an integer value, the RANGE and PATTERN ops apply.
    b. a STRING, for the [char_str] and [alphanum_str], etc. sections. PATTERN ops apply.
    c. a timeval, for all the date/time related sections. All the DATE and time related ops apply. a timeval takes the form of a 64-bit number, in seconds from the 1 Jan 1970 epoch.
3. If an OP is inappropriate for the data passed, it may just evaluate FALSE, but then again, it may not. the parser can notify the user of inappropriate OP usage for the 12 fixed ident values above. For instance, the ODD op would make sense for a time or date related SayScript.

Every line within the body of a language has these five columns:

1. OP -- a comma separated list of operations to perform:
    RANGE
    PATTERN
    GREATER
    LESS
    VAR
    NOTVAR
    NUMLEN
    ATBEGIN
    ATEXIT
    DATEPAST_RANGE
    DATEPAST_GREATER
    DATEFUT_RANGE
    DATEFUT_GREATER
    ANYDATE
    MINUTE_RANGE
    HOUR_RANGE
    YEAR_RANGE
    OPT
    NOTOPT

    If there is more than one OP specified, then the Ops are evaluated in order, and the first OP that evaluates to FALSE terminates the evaluation, and execution proceeds to the next statement. This equates to the ANDing of the multiple OPs.

    a. RANGE. The ARG column should contain two numbers sep by a comma. If we notate the range as "a,b", then the current line is processed if a <= the input number <= b. If the input number is not within this range, the line is skipped, and processing proceeds to the next line.
    b. PATTERN. If the input number matches the regex (enclosed in quotes) in the ARG column, then process the line, otherwise, skip this line and continue with the next line in the script.
    c. GREATER. If the input number is numerically greater than the ARG, then process this line. Skip otherwise.
    d. LESS. If the input number is numerically smaller than the ARG, then process this line. Skip otherwise.
    e. VAR. If the variable name exists and is set to a non-zero value, then process this line, or skip this line.
    f. ATEXIT. Before the script terminates, all statements with ATEXIT will be executed in order. These statements are ignored until that time. They are only evaluated if DONE or RANGE_ERROR is encountered. At that time, the script will be scanned and all ATEXIT lines will be evaluated in sequence. If more than one OP is present, ATEXIT should be FIRST!
    g. DATEPAST_GREATER. Expects one arg, a value in days, If the given date is in the past, greater than the number of days in arg, then process the line.
    h. DATEFUT_GREATER. The analog to DATEPAST_GREATER, but applies to the future.
    i. ANYDATE. Arg is null. Always process this statement.

j. MINUTE_RANGE. Arg is in range format, x <= minutes <= y must be true for statement to be processed.
k. HOUR_RANGE. Like MINUTE_RANGE, only compares the 24 hour time.
l. YEAR_RANGE. Also like MINUTE_RANGE, only for the year.
m. DATEPAST_RANGE. Expects a comma-sep pair of args in days describing x <= timeval <= y; If the day is in the past and within the indicated range, then process the statement.
n. DATEFUT_RANGE. Same as DATEPAST_RANGE, but is true only if the supplied date is in the future, within the number of days specified.
o. NOTVAR. If the variable name has never been set, or is 0, then process this line.
p. OPT. The arg must consist of a single character, or a single character in double-quotes (if the character is not alphabetic.) If it exists in the options passed to the script, then the OP is true.
q. NOTOPT. The arg must consist of a single character, or if the character is not aphabetic, it must be wrapped in double-quotes. If it does NOT exist in the options passed to the script, then the OP is true.
r. ATBEGIN. Before the script is started, all statements with the ATBEGIN op are evaluated in sequence, if any exist. If more than one OP is specified, then the ATBEGIN should be FIRST!
s. SECOND_RANGE. Like MINUTE_RANGE, or HOUR_RANGE, compares the seconds after the minute.

– each of these operations will be explained below.

2. ARG -- the data for the OP(s) to operate on/with:
for RANGE, it will be two numbers, separated by a comma
for PATTERN, it will be a single REGEX expression to match. Must be enclosed in double quotes. use ' \" ' to include a double quote in the regex.
for GREATER, and LESS, it will a single number
for VAR, it will be the name of a user-defined VAR
for ATEXIT, there are no args.
for ATBEGIN, there are no args.
for NUMLEN, a number that will match the length of the input number, or the rule will be skipped.
or, ODD, which will be true if the length of num is an odd number,
or, EVEN, which will be true if the length of num is an even number.
for DATEFORM, the script is run for each format specified in the format. The arg should contain exactly one letter that must match the current format for the statement to be processed. The DATEFORM should only be used in the [date_format] section.
for DATEPAST, if the arg is less than or equal to the difference in the day numbers between the given time, and 'now', then the statement will be processed. You can also say 2W, for 2 weeks, or 2M for 2 months, or 2Y for 2 years, as an abbreviation.
for DATEFUT, the same as DATEPAST, except the input time value will be in the future
for ANYDATE, a ´-´ is acceptable. This is for single rules that cover all dates.
for SECOND_RANGE, a comma separated pair of lower,upper bounds to test for the {time.sec} field.
for MINUTE_RANGE, a comma separated pair of lower,upper bounds to test for the {time.min} field.
for HOUR_RANGE, a comma separated pair of lower,upper bounds to test for the {time.24hour} field.
for YEAR_RANGE, a comma separated pair of lower,upper bounds to test against the {date.year} field.
for DATEPAST_RANGE, a comma-sep pair (x,y) of args in days in the past describing x <= timeval <= y;
all DATEPAST and DATEFUT will calculate this by calculating the time of 00:00:00 on the 24 clock of the current day, and using 86,400 seconds per day from there. 0 days is true if the time passed is between ´nowi, and the beginning of the day.
for DATEFUT_RANGE, the args are the same, and the same heuristics apply, except the date passed to the script must be greater than the current time. 0 days in the future, is a time that is between the beginning of ´today', and before the end of ´today'.
for NOTVAR, a single variable name is the argument, and if it has never been set, or is 0, then this OP is "TRUE"

for OPT, the arg is a char string, wrapped in quotes if the string does not form an Identifier (which must start with an alphabetic character, and may contain only alphanumeric characters).
If any of these chars exist in the option string passed to the script, then This OP is "TRUE" for NOTOPT, to evaluate as "TRUE", none of the chars in the arg string must be in the option string passed to script.

If multiple Ops were specified, then the number of comma separated elements in the ARG list should be the total of the number of args required for each OP. The args for each OP are in the same order as the Ops. The first FALSE OP terminates the evaluation, and the flow of control goes to the next statement. Thus in this statement:

```
OPT,RANGE  f,0,100 ...
```

first, the OPT operation will be tested, to see f is among the options passed, and if it is, then the input number will checked to see if it is between 0 and 100 inclusive. If it is, the statement will be processed, and any sound files will be played.

3. Files to Play: 'notation' separated list of files to play, including expressions. If there are no files to play, a single '-' indicates a null entry.
The 'notation' separating files represents amounts of silence to insert between them. The notation can consist of one or more of these characters:
comma (,); semicolon (;); plus (+); colon (:); period (.), and questionmark (?)
Each separating character represents a single file or set of files containing silence that should be played in that spot. It was intended that the plus (+) should be null silence, so that utterances could be directly joined, and it resemble the concat operator in some languages.

File names may contain the following expressions:

```
{num[x]} -- num[0:0] is the leftmost (first) number in the string. num[1:1] is the next, and so on.
            A single index indicates a substring starting with the specified index.
{num[x:y]] -- x and y form a range; x <= y; x and y are inclusive, so num[0:2] contains the
            first 3 characters of the number/string.
{opt:f} -- will evaluate to 'f' if the option contains the char 'f'. Null otherwise.
{time.sec}  -- 0 to 59; use this in the various date and time sections to be replaced with the
            appropriate field from the time structure.
{time.min}   -- 0 to 59
{time.12hour}   -- 0 to 12
{time.24hour}   -- 0 to 23
{time.12hour2d}   -- 00 to 12
{time.24hour2d}   -- 00 to 23
{time.ampm}     -- "a-m" or "p-m", depending on the 12-hour time.
{time.xm}       --  "am" or "pm" depending on the 12-hour time.
{time.cm}       --  ´A' or ´P' depending on the 12-hour time.
{time.tz}       -- the timezone
{date.dom}      -- 1 to 31
{date.dow}      -- 0 to 6, where 0 is sunday
{date.month}    -- 0 to 11, where 0 is January
{date.dowstr}      -- ´sun', ´mon', ´tues', ´wed', ´thurs', ´fri', ´sat'
{date.monthstr}    -- ´jan', ´feb', ´mar', ´apr', ´may', ´june', ´july', ´aug', ´sep', ´oct', ´nov', ´dec'
{date.year}     -- Full year, at least from 1960 to 2035.
  ALSO, any time: or date: spec can contain range specifiers: {date.year[3]}, {date.year[0:1]}, and etc.
{date.century}  -- result of truncated division of year by 100. Example, for year 1998, century is 19.
{date.decade}   -- the last two digits of the year.
{timeval}       -- the unix timeval passed into the date related funcs. Can be used to call a different
                   Section  (see next item)
<<section:arg>>  -- calls the section (like ´numberi, ´moneyi, etc), and sets the num/date to the arg.
          arg can contain any of the above applicable expressions. Acts like a subroutine call.
```

4. VARSET -- when necessary, a single <varnam> = VALUE expression (with no spaces). A single '-' indicates a null entry.

5. NEXT -- A Single Operation to describe how to move to the next state:

CUT(n) where n == the number of characters to remove from the beginning of the number. RESTART is then performed.

ZERO(n) Set the nth digit from the left to zero, where the leftmost character would be 1. Then, restart. Useful for that out-of-sequence digit, like the 8-and-twenty of German.

RANGE_ERROR says to terminate the script with a range error -- the number input is too large.

RESTART start executing the script again at the beginning

DONE end the script

NEGATE change the sign of the input number, and RESTART

Here's the general rules of this simple language:

1. Execution proceeds in order, from the first line to the last.
2. RESTART operations will force the execution to start at the beginning.
3. DONE terminates the script. The number should have been fully voiced.
4. RANGE_ERROR terminates the script, and the say app should output some error message.

SILENCE INSERTION

The [silence] section must be defined in every SayScript file, to determine the exact length of pause to insert between words or phrases. In this section, each character is assigned a silence file (or set of silence files) to be concatenated in the sound buffer to provide the necessary pause. We can also define different "comma" silences for SaySentence() calls, and script based formats.

These declarations must be defined:

SENT_SPACE = <silence file> (any number of silence files, actually; no spaces or other notation between them, please!)
SENT_COMMA = <silence file> (make sure to include the <> notation to surround each file name)
SCRIPT_COMMA = <silence file>
BOTH_COLON = <silence file>
BOTH_SEMICOLON = <silence file>
BOTH_PERIOD = <silence file>
BOTH_QUESTION = <silence file)
BOTH_PLUS = <silence file> (for null silence, just use a - instead of <> notation)

5. Columns are separated by any number of spaces or tabs.

Example SayScript Files –

SayScripts for the Default Asterisk sound set voiced by Allison

This would be the contents of the file “language/en”:

```
;;; ===========================================================================================================
;;; ===========================================================================================================
;;; ===========================================================================================================
;;; ===========================================================================================================
;;;
;;;;  CURRENT ASTERISK say_number:
;;;
;;;
;;; one of the complications that the current Asterisk English script introduces is
;;; the "delayed thousand/million announcing".
;;; This is common where languages use the "thousand (or any 10^(3x)) to clearly divide the groups of 3 digits.
;;; Thus, 102 "million" 3, is 102,000,003. 102 "million" 3 "thousand" is 102,003,000 & etc.  But since the million
;;; is determined before it is pronounced, we need a method to store and use it later... using the VAR mechanism...
;;;

[silence]    ;; by default, Asterisk with the default sound set, inserts no silence between phrases and words.
SENT_SPACE = -   ;; the silence is expected to be contained in the sound files themselves
SENT_COMMA = -
SCRIPT_COMMA = -
BOTH_COLON = -
BOTH_SEMICOLON = -
BOTH_PERIOD =  -
BOTH_QUESTION =  -
BOTH_PLUS = -


[digit_str]
PATTERN "^[0-9]"  digits/{num[0:0]}                -  CUT(1)
PATTERN "^."      -                                -  CUT(1)


[number]

ATBEGIN,GREATER  999999999    -                      - RANGE_ERROR
ATBEGIN,LESS     0        digits/minus              - NEGATE  ;; turn the num positive and say "minus"

PATTERN  "^0[0-9]"           -                      -  CUT(1)  ;; get rid of leading zeroes!

VAR   SAYMIL             digits/million       SAYMIL=0    RESTART  ;; pronounce the "million" if one of the other rules set the var
VAR   SAYTHOW            digits/thousand      SAYTHOW=0   RESTART  ;; pronounce the "thousand" if one of the other rules set the var

RANGE    0,19            digits/{num}                - DONE
RANGE    20,99           digits/{num[0:0]}0          - CUT(1)
RANGE    100,999         digits/{num[0:0]},digits/hundred   - CUT(1)

;; handle MILLIONS   hundreds/tens/ones, followed by "million"
RANGE 100000000,999999999  digits/{num[0:0]},digits/hundred   SAYMIL=1   CUT(1)
RANGE 20000000,99999999    digits/{num[0:0]}0              SAYMIL=1   CUT(1)
RANGE 10000000,19999999    digits/{num[0:1]}               SAYMIL=1   CUT(2)
RANGE 1000000,9999999      digits/{num[0:0]}               SAYMIL=1   CUT(1)


;; handle THOUSANDS  hundreds/tens/ones, followed by "thousand"
RANGE 100000,999999        digits/{num[0:0]},digits/hundred   SAYTHOW=1   CUT(1)  ;; x "hundred"
RANGE 20000,99999          digits/{num[0:0]}0              SAYTHOW=1   CUT(1)  ;; twenty/thirty/forty/etc
RANGE 10000000,19999999    digits/{num[0:1]}               SAYTHOW=1   CUT(2)  ;; ten thru nineteen
RANGE 1000000,9999999      digits/{num[0:0]}               SAYTHOW=1   CUT(1)  ;; one thru 9

;; For numbers like 102,000, after the "one" "hundred" "two", we'll get 'num' set to zero,
;; and RESTART (always how CUT() ends) will end the script.
;; So, we have to make sure that the "million" or "thousand" gets said before we exit....

ATEXIT,VAR  SAYTHOW          digits/thousand              SAYTHOW=0   DONE ;; only one of these could be set...
ATEXIT,VAR  SAYMIL           digits/million               SAYMIL=0    DONE


[alphanum_str] ;; just a copy of the char_str;; it will say all the alphas and nums, and so much more!
PATTERN  "^[*]"       digits/star                    - CUT(1)
PATTERN  "^[#]"     digits/pound                       - CUT(1)
PATTERN  "^[!]"     letters/exclamation-point              - CUT(1)
```

```
PATTERN  "^[@]"    letters/at                    - CUT(1)
PATTERN  "^[$]"    letters/dollar                - CUT(1)
PATTERN  "^[-]"    letters/dash                  - CUT(1)
PATTERN  "^[.]"    letters/dot                   - CUT(1)
PATTERN  "^[=]"    letters/equals                - CUT(1)
PATTERN  "^[+]"    letters/plus                  - CUT(1)
PATTERN  "^[/]"    letters/slash                 - CUT(1)
PATTERN  "^[ ]"    letters/space                 - CUT(1)
PATTERN  "^[0-9]"  digits/{num[0:0]}             - CUT(1)
PATTERN  "^[a-z]"  letters/{num[0:0]}            - CUT(1)  ;; the code for char_str will downcase all alphabetic
PATTERN  "^."      letters/{num[0:0]}            - CUT(1)  ;; catchall for anything else


[char_str]
PATTERN  "^[*]"    digits/star                   - CUT(1)
PATTERN  "^[#]"    digits/pound                  - CUT(1)
PATTERN  "^[!]"    letters/exclamation-point     - CUT(1)
PATTERN  "^[@]"    letters/at                    - CUT(1)
PATTERN  "^[$]"    letters/dollar                - CUT(1)
PATTERN  "^[-]"    letters/dash                  - CUT(1)
PATTERN  "^[.]"    letters/dot                   - CUT(1)
PATTERN  "^[=]"    letters/equals                - CUT(1)
PATTERN  "^[+]"    letters/plus                  - CUT(1)
PATTERN  "^[/]"    letters/slash                 - CUT(1)
PATTERN  "^[ ]"    letters/space                 - CUT(1)
PATTERN  "^[0-9]"  digits/{num[0:0]}             - CUT(1)
PATTERN  "^[a-z]"  letters/{num[0:0]}            - CUT(1)  ;; the code for char_str will downcase all alphabetic
PATTERN  "^."      letters/{num[0:0]}            - CUT(1)  ;; catchall for anything else


[phonetic_str]
PATTERN  "^[*]"    digits/star                   - CUT(1)
PATTERN  "^[#]"    digits/pound                  - CUT(1)
PATTERN  "^[!]"    letters/exclamation-point     - CUT(1)
PATTERN  "^[@]"    letters/at                    - CUT(1)
PATTERN  "^[$]"    letters/dollar                - CUT(1)
PATTERN  "^[-]"    letters/dash                  - CUT(1)
PATTERN  "^[.]"    letters/dot                   - CUT(1)
PATTERN  "^[=]"    letters/equals                - CUT(1)
PATTERN  "^[+]"    letters/plus                  - CUT(1)
PATTERN  "^[/]"    letters/slash                 - CUT(1)
PATTERN  "^[ ]"    letters/space                 - CUT(1)
PATTERN  "^[0-8]"  digits/{num[0:0]}             - CUT(1)
PATTERN  "^[a-z9]" phonetics/{num[0:0]}_p        - CUT(1)  ;; the code for char_str will downcase all alphabetic
PATTERN  "^."      letters/{num[0:0]}            - CUT(1)  ;; catchall for anything else


[enumeration]
;; PROBLEM: no h-hundred (hundredth), h-thousand (thousandth), h-million (millionth), h-billion (billionth), or h-trillion (trillionth).
;; (which implies, btw, that say_enumeration isn't totally correct in what it generates)
;; [enumeration]  ;; the last number has to finish with 'th', 'st', 'rd', etc, as in 'first', 'second', 'third', 'fourth', etc.
;; to say the 'minus first' is pretty senseless... but so it is.
ATBEGIN,LESS     0             digits/minus     - NEGATE  ;; turn the num positive and say "minus"
ATBEGIN,GREATER  999999999     -                - RANGE_ERROR

PATTERN  "^0[0-9]"  -          -        CUT(1)  ;; get rid of leading zeroes!

VAR  SAYBIL     digits/billion   SAYBIL=0   RESTART  ;; pronounce the "million" if one of the other rules set the var
VAR  SAYMIL     digits/million   SAYMIL=0   RESTART  ;; pronounce the "million" if one of the other rules set the var
VAR  SAYTHOW    digits/thousand  SAYTHOW=0  RESTART  ;; pronounce the "thousand" if one of the other rules set the var


RANGE    0,19          digits/h-{num}       - DONE
PATTERN  "^[2-9]0"     digits/h-{num}       - DONE  ;; tenth, twentieth, thirtieth, fourtieth, etc.
RANGE    20,99         digits/{num[0:0]}0   - CUT(1)  ;; twenty, thirty, etc, followed by 'first', 'second', etc,
;;                                                    via the previous rule after the restart
PATTERN  "^[1-9]00"    digits/{num[0:0]},digits/h-hundred - DONE  ;; three hundredth car... Get the even hundredths with this one.
RANGE    100,999       digits/{num[0:0]},digits/hundred   - CUT(1)

;; handle BILLIONS   hundreds/tens/ones, followed by "billion"
RANGE 100000000000,999999999999  digits/{num[0:0]},digits/hundred  SAYBIL=1   CUT(1)
RANGE 20000000,99999999    digits/{num[0:0]}0                SAYBIL=1   CUT(1)
RANGE 10000000,19999999    digits/{num[0:1]}                 SAYBIL=1   CUT(2)
RANGE 1000000,9999999      digits/{num[0:0]}                 SAYBIL=1   CUT(1)


;; handle MILLIONS   hundreds/tens/ones, followed by "million"
RANGE 100000000,999999999  digits/{num[0:0]},digits/hundred   SAYMIL=1   CUT(1)
RANGE 20000000,99999999    digits/{num[0:0]}0                 SAYMIL=1   CUT(1)
RANGE 10000000,19999999    digits/{num[0:1]}                  SAYMIL=1   CUT(2)
RANGE 1000000,9999999      digits/{num[0:0]}                  SAYMIL=1   CUT(1)
```

```
;; handle THOUSANDS  hundreds/tens/ones, followed by "thousand"
RANGE 100000,999999       digits/{num[0:0]},digits/hundred   SAYTHOW=1   CUT(1)  ;; x "hundred"
RANGE 20000,99999         digits/{num[0:0]}0                 SAYTHOW=1   CUT(1)  ;; twenty/thirty/forty/etc
RANGE 10000000,19999999   digits/{num[0:1]}                  SAYTHOW=1   CUT(2)  ;; ten thru nineteen
RANGE 1000000,9999999     digits/{num[0:0]}                  SAYTHOW=1   CUT(1)  ;; one thru 9

;; For numbers like 102,000, after the "one" "hundred" "two", we'll get 'num' set to zero,
;; and RESTART (always how CUT() ends) will end the script.
;; So, we have to make sure that the "million" or "thousand" or whatever gets said before we exit....

;; BTW, the current Asterisk implementation of enumeration is incorrect...
;; it doesn't use h-hundred, h-thousand, h-million, or h-billion...  you need
;; to use these to say stuff like "that was the 161-thousandth cat..."
;; NOT "that was the 161st thousand cat' or like nonsense.

ATEXIT,VAR  SAYTHOW          digits/h-thousand            SAYTHOW=0   DONE ;; only one of these could be set... "thousandth"
ATEXIT,VAR  SAYMIL           digits/h-million             SAYMIL=0    DONE  ;; "millionth"
ATEXIT,VAR  SAYBIL           digits/h-billion             SAYBIL=0    DONE  ;; "billionth"




[date]
;; by the way, the current Asterisk of say_date is a little broken,
;; it uses say_number for the day-of-month number, so it will say
;; something like "Friday, June twenty-three, 2009", when it would
;; probably be better to say "Friday, June twenty-third, 2009"
ANYDATE  -  digits/day-{date.dow},digits/mon-{date.month},<<enumeration:{date.dom}>>,<<number:{date.year}>>  - DONE


[time]
;; this should say something like "10 oh four pm" or "12 twenty two am" or "11 oclock am"
MINUTE_RANGE 0,0   <<number:{time.12hour}>>,digits/oclock,digits/{time.ampm}                -  DONE
MINUTE_RANGE 1,9   <<number:{time.12hour}>>,digits/oh,digits/{time.min},digits/{time.ampm}   -  DONE
MINUTE_RANGE 10,59 <<number:{time.12hour}>>,<<number:{time.min}>>,digits/{time.ampm}         -  DONE

[date_time]
ANYDATE  -  <<date:{timeval}>><<time:{timeval}>>          - DONE             ;; XXX for both */


;; this is a little different, in that the formatted date pronouncer has a rather rich set of
;; operators, that pronounce different items in the order specified. So, the say_date_format
;; routine will call the date_format_<char> for each item in the format list; if a file is in the mix,
;; the file is played in sequence a level above the the sayscript. The sayscript is only concerned
;; with the format specs, and pronouncing the expected item correctly.

[date_format_A,date_format_a]
;; "Sunday", "Monday", etc
ANYDATE - digits/day-{date.dow}       - DONE


;; "January", "February", etc
[date_format_B,date_format_b,date_format_h]
ANYDATE - digits/mon-{date.month}       - DONE

;; "first', "second', ... "thirty first'  day of month
[date_format_D,date_format_e]
ANYDATE - <<enumeration:{date.dom}>>  - DONE


;; Year "two thousand 10", etc
[date_format_Y]
YEAR_PATTERN "^1[0-9]0[1-9]"    <<number:{date.century}>>,digits/oh,digits/{date.year[3]}     - DONE
YEAR_PATTERN "^1[0-9]00"    <<number:{date.century}>>,digits/hundred                          - DONE
YEAR_RANGE  0,1999  <<number:{date.century}>>,<<number:{date.decade}>>                        - DONE
YEAR_RANGE  2000,2099  <<number:{date.year}>>                                                 - DONE
YEAR_PATTERN "^[2-9][0-9]0[1-9]"   <<number:{date.century}>>,digits/oh,digits/{date.year[3]}  - DONE
YEAR_PATTERN "^2[1-9]00"    <<number:{date.century}>>,digits/hundred                          - DONE
YEAR_RANGE  2100,9999  <<number:{date.century}>>,<<number:{date.decade}>>                     - DONE

;; hour in 12-hour format "oh' to 12
[date_format_I,date_format_l]
HOUR_RANGE 0,0  digits/oh                    - DONE
HOUR_RANGE 1,23 <<number:{time.12hour}>>       - DONE


;; hour in 24-hour format H as oh-8, k just says 8;
[date_format_H]
HOUR_RANGE 0,0  digits/oh                      - DONE
HOUR_RANGE 1,9  digits/oh,<<number:{time.24hour}>>  - DONE
HOUR_RANGE 10,23 <<number:{time.24hour}>>          - DONE
```

```
[date_format_k]
HOUR_RANGE 0,0  digits/oh                - DONE
HOUR_RANGE 1,23  <<number:{time.24hour}>>  - DONE


;; month number, ¢first', ¢second', ..., ¢twelfth'  ;; who on earth will use this... oh, well!
[date_format_m]
ANYDATE -  <<enumeration:{date.month}>>  - DONE


;; say the minutes, M says ¢oclock¢ if 0; N says ¢hundred¢ if zero. ¢oh¢ before mins less than 10;
[date_format_M]
MINUTE_RANGE   0,0    digits/oclock                - DONE
MINUTE_RANGE   1,9    digits/oh,<<number:{time.min}>>  - DONE
MINUTE_RANGE   10,59  <<number:{time.min}>>          - DONE

[date_format_N]
MINUTE_RANGE   0,0    digits/hundred               - DONE
MINUTE_RANGE   1,9    digits/oh,<<number:{time.min}>>  - DONE
MINUTE_RANGE   10,59  <<number:{time.min}>>          - DONE


;; AM/PM
[date_format_p,date_format_P]
ANYDATE -   digits/{time.ampm}                     - DONE


;; Relative date; ¢today¢(for Q), (nothing for today for q)

[date_format_q]
DATEPAST_RANGE  0,0   -                                              - DONE  ;; today
DATEPAST_RANGE  1,1   digits/yesterday                                 - DONE  ;; yesterday
DATEPAST_RANGE  2,6   <<date_format_A:{timeval}>>                      - DONE  ;; w/l 6 days
DATEPAST_RANGE  7,30  <<date_format_A:{timeval}>>,<<date_format_B:{timeval}>>,<<date_format_d:{timeval}>> - DONE
DATEPAST_RANGE  31,183 <<date_format_B:{timeval}>>,<<date_format_d:{timeval}>>          - DONE
DATEPAST_GREATER  183  <<date_format_B:{timeval}>>,<<date_format_d:{timeval}>>,<<date_format_Y:{timeval}>> - DONE
DATEFUT_RANGE   1,1   digits/tomorrow                                  - DONE ;; an enhancement!
DATEFUT_GREATER  1    <<date_format_B:{timeval}>>,<<date_format_d:{timeval}>>,<<date_format_Y:{timeval}>> - DONE


[date_format_Q]
DATEPAST_RANGE  0,0   digits/today                                   - DONE  ;; today
DATEPAST_RANGE  1,1   digits/yesterday                                 - DONE  ;; yesterday
DATEPAST_RANGE  2,6   <<date_format_A:{timeval}>>                      - DONE  ;; w/l 6 days
DATEPAST_RANGE  7,30  <<date_format_A:{timeval}>>,<<date_format_B:{timeval}>>,<<date_format_d:{timeval}>> - DONE
DATEPAST_RANGE  31,183 <<date_format_B:{timeval}>>,<<date_format_d:{timeval}>>          - DONE
DATEPAST_GREATER  183  <<date_format_B:{timeval}>>,<<date_format_d:{timeval}>>,<<date_format_Y:{timeval}>> - DONE
DATEFUT_RANGE   1,1   digits/tomorrow                                  - DONE ;; an enhancement!
DATEFUT_GREATER  1    <<date_format_B:{timeval}>>,<<date_format_d:{timeval}>>,<<date_format_Y:{timeval}>> - DONE

;; Equiv of HM,  ¢oh¢¢eight¢¢thirty¢¢three¢
[date_format_R]
ANYDATE - <<date_format_H:{timeval}>>,<<date_format_M:{timeval}>>  - DONE


;; Seconds ¢zero¢, ¢oh¢ five, and so on
[date_format_S]
SECOND_RANGE  0,0  digits/0                  - DONE
SECOND_RANGE  1,9  digits/oh,<<number:{time.sec}>>  - DONE
SECOND_RANGE  10,59  <<number:{time.sec}>>          - DONE

;; T is the equiv of HMS
[date_format_T]
ANYDATE - <<date_format_H:{timeval}>>,<<date_format_M:{timeval}>><<date_format_S:{timeval}>>  - DONE



[rel_date_time]  ;; say_datetime_from_now
;; We can easily extend this into the future with ¢today¢ and a little wave of the wand!
DATEPAST_RANGE  0,0   <<time:{timeval}>>       - DONE  ;; today: just tell the time                  ;; XXX */
DATEPAST_GREATER  6   digits/mon-{date.month},<<enumeration:{date.dom}>>,<<time:{timeval}>>   - DONE     ;; XXX */
DATEPAST_RANGE  1,6   digits/day-{date.dow},<<time:{timeval}>>                 - DONE       ;; XXX */
DATEFUT_GREATER   0   digits/mon-{date.month},<<enumeration:{date.dom}>>,<<time:{timeval}>>   - DONE     ;; XXX */
DATEFUT_RANGE   0,0   <<time:{timeval}>>       - DONE  ;; today: just tell the time                  ;; XXX */




[money]
;; the string fed to say_money should be in the format xxxx.xx   -- always!
```

```
;; for the sake of RANGE, the .xx is removed before calculation.
;; using a say_money is smart for translation purposes. It might help
;; fix the gender and other variables that would affect reading off the numbers,
;; allows you to use the local currency name, etc.
ATBEGIN,GREATER   999999999          -       - RANGE_ERROR

PATTERN         "^0[0-9]"          -       -  CUT(1)   ;; leading zero removal

VAR   SAYMIL              digits/million                SAYMIL=0   RESTART  ;; pronounce the "million" if one of the other rules set the var
VAR   SAYTHOW             digits/thousand               SAYTHOW=0   RESTART  ;; pronounce the "thousand" if one of the other rules set the var

;; use the .xx to read off the cents
;; btw, ¢ents is in the extras, which is funny... Seems pretty basic to me
PATTERN "^[.]00"            digits/dollars,vm-and,digits/0,cents                    - DONE
PATTERN "^[.][0-9]0"        digits/dollars,vm-and,digits/{num[1:1]}0,cents              - DONE
PATTERN "^[.]0[1-9]"        digits/dollars,vm-and,digits/{num[2:2]},cents               - DONE
PATTERN "^[.]1[1-9]"        digits/dollars,vm-and,digits/{num[1:2]},cents             - DONE
PATTERN "^[.][2-9][1-9]"    digits/dollars,vm-and,digits/{num[1:1]}0,digits/{num[2:2]},cents - DONE

RANGE    0,19             digits/{num}                        - DONE
RANGE    20,99            digits/{num[0:0]}0                    - CUT(1)
RANGE    100,999          digits/{num[0:0]},digits/hundred          - CUT(1)

;; handle MILLIONS   hundreds/tens/ones, followed by "million"
RANGE 100000000,999999999  digits/{num[0:0]},digits/hundred   SAYMIL=1   CUT(1)
RANGE 20000000,99999999    digits/{num[0:0]}0              SAYMIL=1   CUT(1)
RANGE 10000000,19999999    digits/{num[0:1]}             SAYMIL=1   CUT(2)
RANGE 1000000,9999999      digits/{num[0:0]}             SAYMIL=1   CUT(1)

;; handle THOUSANDS  hundreds/tens/ones, followed by "thousand"
RANGE 100000,999999        digits/{num[0:0]},digits/hundred  SAYTHOW=1   CUT(1)  ;; x "hundred"
RANGE 20000,99999          digits/{num[0:0]}0              SAYTHOW=1   CUT(1)  ;; twenty/thirty/forty/etc
RANGE 10000000,19999999    digits/{num[0:1]}             SAYTHOW=1   CUT(2)  ;; ten thru nineteen
RANGE 1000000,9999999      digits/{num[0:0]}             SAYTHOW=1   CUT(1)  ;; one thru 9


;; we won't need the ATEXIT stuff like say_number, because the .xx at the end will insure the
;; script doesn't exit early and not say what it needs to.
```

Writing a SayScript to Include Gender

To write, say, a "number" SayScript that pays attention to gender, you would include the gender as an option in the SaySentence call. You should be able to tell which gender should be applied, based on the context in which the call is made.

For instance, if in English, the SaySentence call uses this format string:

"<She_is> %n <years_old>."

Let's assume that you would use "n" for neuter, "f" for female gender, and "m" for male.

Then, in the translation file for your language, you'd say:

["<She_is> %n <years_old>."]
"<Sie_ist> %[f]n <Jahre_alt>"

(Yes, the above looks like German. Ignore that!)

The "number" SayScript would then use the OPT:n as a conditional for neuter behavior, OPT:f for female behavior, and OPT:m for male behavior.

What We Offer

- **Working SayScript Interpreter and Translation Engine In Java**
- **A Large Number of Asterisk Code Conversions to SaySentence()**
- **Prototype SaySentence() func and app that uses the Java Engines.**
- **Basic SayScripts for English in the "minimal sound set" style. Tested. Complete.**

Asterisk Implementation Options

– Quick, easy, very incomplete

In this option, we use the external Java server to do the work of SaySentence. We leave Asterisk as it is, but allow IVR's and apps to use the SaySentence() machinery, ExternalVR and the agi() facilities should SaySentence() capabilities. SoundPacks will only be useful for IVR's and such, but hey, at least IVR designers will have some cool stuff to help them with internationalization!

– Complete Implementation

For this option, we excise all the stream-file usages, and convert them to use SaySentence() instead.

With this done, Asterisk in its entirety can be front-ended with Sound Packs. New languages can be generated for Asterisk without any C-code development, testing, proposals, or process.

I personally suggest wrapping all "old" code in #ifdef constructs, with SaySentence calls in the #else half (or vice-versa), and letting the two versions co-exist until the SaySentence() half is solid and complete, then obsolesce/remove the old say_* code.

– SayScript and Translation engines – Internal or External

While the Java SaySentence code will suffice to allow SaySentence to do its thing, there will most likely always be a desire to have that run internally from a .so module. This can and probably will be done.

Some Thoughts

In the course of working on converting code to use ast_say_sentence(), I have bumped into some interesting usages. One of these is using config files to play sounds. In such cases, I advise that these entries be removed from the config file entirely, and a fixed file be played via say_sentence(). If anyone desires to play a different file, they can just replace the file with a different sound file bearing the same name, OR, they can use a translation file to map from one file to another....

General Rules of Coding SaySentence():

1. Say whole sentences at a time. Paragraphs are OK, too.
2. Phrases (files) should not be stored in a variable. If you have 3 possibilities of what to say, then have three SaySentence() calls.
3. Don't code in alternates depending on a number. You can/should use the translation files for this. (eg. "You have 0 boxes" vs. "You have 1 box" vs "You have 2 boxes"; just code "You have %n boxes", and use the translation file to specifiy the different permutations.)
4. In general, follow the gettext rules!