Foreign Language Support

Revision 10/15/2023 c13.4

The set of semi-supported languages will always be in the folder LIVEDATA, as each language there at least recognizes numbers and some other things.

German has additional spellcheck support. Spanish has inbuilt pos-tagging and spellcheck support. Filipino has additional spellcheck support.

Emoji

OK. So it's not really a foreign language. But it does have unique support. First off, in LIVEDATA/interjections.txt you can see some emoji declarations like:

```
:( ~emosad EMOJI
```

This goes one step beyond saying that :(should mark the concept ~emosad. It also declares it as an emoji. All declared emoji are members of ~emoji and ~interjections. And, among other things, if you jam a bunch of emoji together in input, the tokenizer will separate them into distinct emoji that can be recognized. And you can define a concept of emoticons via

```
concept: ~stuff EMOTICON ( ... )
```

Just be careful to keep a space between the last emoticon and the closing).

Foreign Language Overview

ChatScript comes natively with full English support. If you want to use a different language you need a variety of things. * Pos-tagging support (optional if you don't use pos-based keywords) * Spell check support * Concepts in the language (that you use) * LIVEDATA substitutions appropriate to the language (including month names, number names, currencies, plurals * Patterns in the language * Output in the language * Lemmas (comes with foreign pos-taggers or available from ^pos(canonical))

ChatScript has a command line parameter `language=` that tells CS the language you intend. It defaults to `ENGLISH`. The effects of this parameter are generically these, unless special information is given below about the language: * If not ENGLISH, internal pos-tagging (other than marking possible english tags and numbers and dates and such) and parsing are disabled. * If treetagger is licensed and has that language, it will pos tag. * The system will use DICT/`language`. * The system will use LIVEDATA/`language` * The script compiler will automatically compile lines marked to be conditionally compiled with the language (see language comments). * The system will store the user's topic file suffixed by language as well.

While concepts, topics, and functions can be owned by a bot, only words and facts have a language affiliation. Concepts, topics, functions, variables, etc are all commonly visible in all languages, having a language affiliation of UNIVERSAL. You can declare words to have language UNIVERSAL, in which case all other language copies of the word disappear and the facts based on them transferred to the UNIVERSAL form. Facts can never mix two languages (except where UNIVERSAL is one of them) and take on the language of any field that is language constrained.

Embedded Foreign Language Support

Using a language generally means using the dictionary of that language and spell-checking in that language. It may also control recognition of the numeric value of words (like dozen == 12), knowing the names of the months and currencies. ChatScript is told what languages to make available using the command line parameter "language=" . If you don't supply this, the default language is English. But you can for example say "language=german".

You can name up to seven different languages (for which you have dictionaries) by listing them on that parameter "language=english,german,spanish,japanese". You need to use the same language sequence for all compilations and normal execution (hence its a parameter typically listed in cs_init.txt). The first language listed is always the default language (and if you have english, best to make that first).

TREETAGGER POS TAGGING

Iff you have Treetagger licenses, you can use treetagger by putting on a cs_init line something like:

```
treetagger=english,spanish
```

Treetagger improves CS's inbuilt pos-tagging, and provides the only pos-tagging for german and various other languages. To access treetagger, you need additional library files from me. Write me with copies of your treetagger licenses.

If you currently have treetagger for CS, the latest feature allows you to supplement the inbuilt vocabulary of a language. You need to put into your treetagger directory a file like "spanish.pos" which lists each new word on a line with appropriate data. E.g. for spanish:

```
chatear     VLinf   chatear
chateo      VLfin   chatear     NC  chateo
```

This lists the word, its POS tag, its lemma, and additional POS tag and lemma pairs as appropriate. You need to consult the DICT documentation of tags for

that language to know how to label the POS. Likely you should email me and have me help you.

`:language xxx`

allows you to change language on the fly in standalone mode.

All API external functions(^compilepattern, ^testpattern, ^compileoutput, ^testoutput) accept a top-level "language" parameter which sets the language for the duration of the call. The script compiler can be told "language: spanish" at the top level of a file, to change language for compilation for future script. During local execution you can type the debug command ":language german" to change the language recognized by the bot. You can use ^setlanguage(english) from script to change language. At the end of every volley, the server returns to the default language. But for any conversation, the language it was last in is memorized and that user's conversation will resume next volley in that language.

The current language implies using the dictionary corresponding to that language and using spell-checking rules devoted to that language as well as numeric conversions of words in that language.

If `language=ideographic` is used, then spell check is disabled and tokenization will make each character be a token. This is useful for languages like Korean and Chinese.

Words & Facts

When you load multiple dictionaries, you can have multiple copies of the same word. Each exists independently and when a word is looked up, it is mated to the word found in that language. Variables, concept names, and function names are not in the dictionaries that were loaded, and are considered universal names. Facts that get created are also specific to a language, so facts are flagged with what language they come from and when you retrieve any of its fields, they will be decoded from the language of the fact. This means that while a concept name is visible in any language, its members are specific to that language. Two english words are considered universal words, and they are not listed in foreign dictionaries. (portugese). These are used for system facts relating to concept membership and english dictionary hierarchy.

When you build, you can specify what language is to be used for some section of your files. This means that concepts, topics, and facts created will be accessible only from that language. There is, however, the language UNIVERSAL you can specify, which intends that the facts, topics, and words created from that will be accessible to all languages. Note the phrase "words created". Suppose we want to create a word "bacck" in universal. But it already exists for some language. If we use that word, it is not visible to other languages. But if we create a new word entry for the universal language, we now hide the other entries because universal word is found first and is acceptable for all languages. We will lose seeing definitions of english words from Wordnet. And pos-tags for them.

Japanese

ChatScript uses the std CS engine, has no special dictionary or LIVEDATA for Japanese. It knows something is japanese because either the a command line parameter has "language=japanese", or because japanese characters are being used in compiling a pattern (^compilepattern), or because a global variable is passed in via ^testpattern "$language = japanese".

^testpattern will change bits of japanese-written punctuation in a pattern to english to be compatible with pattern matching. This includes japanese ( ) [ ] { } space. All variables need . - _ letters and digits converted from japanese to ascii.

^testoutput will remove all spaces not within doublequotes. And will change bits of japanese-written punctuation in a pattern to english to be compatible. This includes japanese ( ) [ ] { } space. All variables need . - _ letters and digits converted from japanese to ascii.

Mecab Japanese tokenizer

CS uses the Mecab open source library to tokenize a continuous stream of Japanese characters, to provide the pos-tags and lemmas. To use Japanse you need to go to common.h and remove #define DISABLE_JAPANESE 1 and install mecab on your machines. Here are the instructions for that...

Here is how to install and build the mecab Japanese tokenizer libraries used by ChatScript. These instructions are based on the web page here.

```
Windows - prebuilt library  (THIS HAS ALREADY BEEN DONE FOR YOU)
install visual studio
go to https://taku910.github.io/mecab/#download
download mecab-0.996.exe
chrome will warn you, click "Download anyway"
open the executable
set the char set to utf-8
accept the license
put in the default location
install
permit all users
allow compilation
finish
copy the library from C:\Program Files (x86)\MeCab\bin\libmecab.dll to Pearl\BINARIES

Linux (on Ubuntu) - Install using apt-get
sudo apt-get install -y mecab mecab-ipadic-utf8 libmecab-dev

Linux (on Centos 7)
Before starting the install, run: yum install gcc-c++
```

```
go to a-file1/Common/Install\ Software/MeCab to get the installation files
the files came from https://taku910.github.io/mecab/#download
convert to English using Google Translate, if necessary, to read the page. (smile)
copy the mecab source locally: mecab-0.996.tar.gz
unzip the source: tar xfz mecab-0.996.tar.gz
cd mecab-0.996
chmod -R 0755 .
./configure
make
sudo make install
cd ..
copy the ipa dictionary locally: mecab-ipadic-2.7.0-20070801.tar.gz
unzip the dictionary: tar xfz mecab-ipadic-2.7.0-20070801.tar.gz
cd mecab-ipadic-2.7.0-20070801
chmod -R 0755 .
sh ./configure --with-charset = utf8
make
sudo make install
cd ..
copy the static library up: cp mecab-0.996/src/.libs/libmecab.a .
echo "/usr/local/lib" >> /etc/ld.so.conf
(you may have to do echo after first write permitting the destination file)
ldconfig
```

###Numbers

For english numbers as digits residing adjacent to japanese characters, the english number is broken off intact. Thus ??72????? becomes ? ? 72 ? ? ? ? ? on inputs

Sentence Terminators

The major japanese characters acting as sentence terminators (the equivalent of ?, ! and .) in user inputs are automatically converted to their ascii equivalents so that CS rules regarding that punctuation will be maintained for pattern matching or rule invocation.

GERMAN

The spell checker has code to break apart an unknown compound word into its separate recognizable pieces, based on https://www.dartmouth.edu/~deutsch/Grammatik/Wortbildung/Komposita.
. So “Esszimmer” (dining room) becomes Ess Zimmer.

CS comes with a german dictionary, and spell checking will use it. In particular, if the input lacks appropriate accent marks, CS will likely fill them in for you.

New postag attribute labels for German: ~spanish_she, ~german_he, ~german_neuter, german_accusative, ~german_dative, ~german_nominative, ~ger-

man_genitive_object_complement

SPANISH

The tokenizer will simply delete any upside down question or exclamation marks.

CS comes with a spanish dictionary, and spell checking will use it. In particular, if the input lacks appropriate accent marks, CS will likely fill them in for you.

CS can use inbuilt rules for adjectives and nouns to mark a word with plurality (~spanish_singular or ~spanish_plural) and gender (~spanish_he or ~spanish_she). For verbs, CS recognizes present, past, and future tense (simple, marked as ~spanish_future) as well as ~verb_imperative.

Pronouns will be marked with ~pronoun_object_singular or ~pronoun_object_plural or ~pronoun_object_you. Also ~pronoun_indirectobject_singular and ~pronoun_indirectobject_plural and ~pronoun_indirectobject_you. Also ~pronoun_I and ~pronoun_you.

MULTIPLE LANGUAGE DICTIONARY

ChatScript leans heavily on its dictionary, which normally is in a single language. But you can support up to 3 languages in the dictionary simultanously as well as the "universal" language. Words and facts are segregated by language and are only visible to a matching current language.

The command line parameter `language=` can consist of a series of languages separated by commas. This enables multi-dictionary behavior and with it, the ability to change the current language on the fly. E.g., `language=english,spanish,german,japanese` Note that japanese involves no dictionary at all, so it can be listed last without compromising the 3-language limit.

Variables, concept and topic names, numbers, operators and punctuation are language agnostic and always visible.

LEVEL 0

The script compiler has `language: xxx` as a construct to allow you to mix compilation of data in various languages. One can write the files0.txt file to provide data for multiple languages like this:

```
RAWDATA/ONTOLOGY/ENGLISH//
RAWDATA/WORLDDATA/

language: GERMAN
RAWDATA/ONTOLOGY/GERMAN//

language: SPANISH
```

`RAWDATA/ONTOLOGY/SPANISH//`

Scripts can change language on the fly using ^language and testing supports the :language command to change language on the fly from user input. These changes only apply to the current volley.

INPUT and OUTPUT

ChatScript supports UTF8, so making output or patterns in the language is entirely up to you. Ditto for LIVEDATA.

ChatScript supports two kinds of conditional compile comments. Single line comments look like this:

```
#ENGLISH this line will compile if the language is English.
#GERMAN this line will not compile if the language is English.
```

As always, such comments run til end of line. The other comment is the block comment like this:

```
##<<ENGLISH these lines will be compiled under English
until a normal closing block comment ##>>
```

Using conditional compilation, you can make English and other language versions of code sit side by side if you want to.

DICTIONARY

The dictionary file can be just a list of words of the language, one per line. You must list all conjugations of a word because there is no in-built support to figure that out. You may also add english equivalent pos tags (see examples in existing foreign language dictionaries) if you want to use existing keywords tied to pos-tags.

In addition to normal words, there is a file LIVEDATA/. . . /numbers.txt that for a language describes a number word and what it's implied number meaning is.

`:buildforeign language` can be used to rebuild a foreign dictionary given the rawwords data in TreeTagger directory (which you dont have) and

POS-TAGS AND LEMMAS

If you want actual POS values and lemmas (canonical form of a word), you will need a POS-tagger of some sort or use ^pos(canonical) ona word. While it is possible to hook in an external tagger via a web call, that will be noticably slower than an in-built system. You would call the service and then appropriately decode its output using^setcanon, ^settag, ^setrole (if you get such from external service), and ^setoriginal (maybe).

ChatScript supports in-build TreeTagger system, which supports a number of languages. However, you can only use this if you have a commercial license. You can try it out using `^popen`, as is done in the German bot, however it will be slow because it has to reinitialize TreeTagger for every sentence. The in-built system does not. A license (per language) is about $1000 for universal life-time use. You can contact me if you want to arrange to use it.

Ontology

CS ships with a Spanish and some other dictionaries that provides spelling of words (for spell correction) and parts of speech of words. It also ships with some ontologies like LIVEDATA/ONTOLOGY/SPANISH which you can do :build 0 if you have set language=SPANISH in cs_init.txt file.

Translating scripts

There is built-in code to translate scripts using Microsoft Translate. It requires you have an api key for it (but you can sign up for free and a bunch of credit. The scripts are translated as follows:

```
1. topic keywords are translated individually
2. pattern keywords that are not number or concept names or variable names are translated ir
3. Output sentences are translated intact for sections between computations.
4. #! sample inputs are translated
```

The original text is put in the output file in comments, along with the translation. This allows human translators to verify the translations. The arguments to this capability are:

```
:translatetop filename original_language new_language
```

The file will be written to tmp/filename. Languages use the MS/utf8 language naming conventions where english is en_US, german is de_DE, french canadian is fr_CA, and french french is fr_FR. If your topic name or concept names or rule labels seen have the name of language in them, they are translated to new name. Eg

```
# -- topic: ~QUIBBLE_en_US keep repeat nostay (beer)
topic: ~QUIBBLE_de_DE keep repeat nostay (Bier)
# -- #! awesome
# #! Prima
# -- u: QUIBBLE_en_US(<<[awesome great wonderful]>>) Agreed.
u: QUIBBLE_de_DE(<<[prima Großartig wunderbar]>>)
    Einverstanden.
```

Translating concepts

There is built-in code to translate concepts using Google Translate. It requires you have an api key for Google Translate (but you can sign up for free and get $300 worth of credit good for 3 months which is enough to do all your translation work probably). You tell CS this as a command line parameter:

```
apikey=AIzaSyAxxxx
```

When I want to translate all level 0 concepts I do the following:

1. erase the contents of TOPIC folder
2. `:build 0`
3. run CS using command line parameter `noboot` and your apikey
4. `:sortconcept x`
5. `:translateconcept german myfilename`

If you run ^csboot and that generates new concept data then you need `noboot`, otherwise it doesn't matter.

`:sortconcept x` locates all currently defined concepts (hence just :build 0) and writes them out to a top level file named `concepts.top` with one concept per line. This file will be read by the next stage.

`:translateconcept` uses the apikey. It reads each line of concepts.top (1 line per concept) and calls google translate for the language you named, saving the results in the path/file you gave. Currently this only recognizes the following language names: german, french, italian, spanish, russian, hindi. I could add more if needed.

The resulting file will automatically prepend each line with conditional compile markers for the language you named, so you can directly add it to your bot and it will only compile when you are in that language mode.

If you want to translate concepts from your bot, then do the following:

1. erase the contents of TOPIC folder
2. `:build harry` (or whatever your bot is)
3. run CS using command line parameter `noboot` and your apikey
4. `:sortconcept x`
5. `:translateconcept french myfilename`

If you just want to translate a single concept/topic then you can call

```
:translateconcept ~myconcept french myfilename
```

It will, as a byproduct, provide the sorted english form of the concept on a single line in `cset.txt`. If you dont give a language and filename, then it will just sort your english concept and write it out.