Foreign Language Support

Revision 11/21/2021 cs11.6

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.

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 three different languages (for which you have dictionaries) by listing them on that parameter "language=english,german,spanish". You need to use the same language sequence for all compilations and normal execution (hence its a parameter typically listed in cs_init.txt). You can even add "japanese" at the end of your list (not earlier) because that requires no dictionary. The first language listed is always the default language.

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.

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”.

CompilePattern

ChatScript directly supports Japanese coding in ^compilepattern. A “word” is a sequence of characters with no spaces in it (except for double quoted strings). Note in the texts below, ? stands for some japanese character.

Patterns that are compiled and that have tokens of Japanese UTF8 characters will expand those tokens to put spaces between each character. Each character thus becomes a word internally.

So this, (_??? $var:=_0), which a scripter writes as a single word to be found and memorized, internally is sent to cs as this:

[{“dse”: {“compilepattern”: “(_??? $var:=_0)" }}]

and CS returns this for internal use, where each character is now a separate word: [{ “dse”: { “testpattern”: {input: " ???? ?¿‘,”patterns“: [{ “pattern”: “^(_(? ? ?) :6$var:=_0) " }] } } }]

This pattern says that the given character sequence MUST be detected as a contiguous sequence. You can use the usual exclusion and memorization operators on such words, and they will be applied to the sequence.

^testpattern

If you set a global variable “language” to “japanese”, then in calls to ^testpattern (the condition of a node) will adjust user inputs that have japanese characters in the following ways:

Normal character clusters are separated into discrete characters. These can match the patterns that have been compiled. English AND Japanese terminal punctuation marks corresponding to period, question , and exclamation are internally converted to English (because they will be stripped off the sentence as per usual). If the pattern matcher is required to memorize a “word”, it will find the separated characters and join them back together to return the value. Thus the above testpattern call will return:“output”: {“newglobals”: {“$var”: “???”} , “match”: 0} }]
Concepts are written per usual, using Japanese words (characters without spaces).

Sentence Limit

CS has a limit of 254 words per sentence, which in this context means 254 characters per sentence of input. Where terminal punctuation is detected, that ends a sentence. Where the limit is reached before finding any terminal punctuation, CS does what it does with English, it breaks the sentence at that point and remaining words will be allocated to the next sentence. Ryan ran some numbers on about 100,000 user inputs from our jp bots. The average character length was 33, and inputs greater than 254 characters only account for 0.5% of all volleys.

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.

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.

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.

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 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.