1 Module model

```
Bachelor AI project 2009. Andreas van Cranenburgh 0440949.
Two-word stage dialogue simulator using a corpus of exemplars.

Interactive usage:
$ python model.py

Other uses:
- Re-enact dialogue in 01.cha (should be a text file with one utterance per
  line, with each line preceeded by *MOT or *CHI, as in the CHILDES corpus):

  $ python
  >>> import model
  >>> model.evaluate(open('01.cha').read())
  [...]

- Talk to self for 25 utterances:

  >>> model.selftalk(25)
  [...]
```

1.1 Functions

arg(*token*)

```
>>> arg('foo(bar)')
'(bar)'
```

conciliate(*meaning*, *partialmeaning*, *subst*=None, *debug*=<function dbg at 0x84b35a4>)

Test whether meaning can be made comptabible with partialmeaning, by looking for a family resemblance with partialmeanings, if found, perform a substitution if necessary.

```
>>> partialmeaning = ['assertion: point(dog) animal(dog)']
>>> conciliate('question: animal(bunny) do(X)', partialmeaning)
substituted (dog) with (bunny)
True
>>> partialmeaning
['assertion: point(bunny) animal(bunny)']
>>> conciliate('assertion: do(hop) animal(bunny)',
 ['assertion: animal(bunny) do(X)'])
instantiated (X) with (hop)
True
>>> conciliate('assertion: do(hop) animal(bunny)',
 ['assertion: point(bunny)'])
False
```

dbg(**m*)

Print debug output (also see nodbg). Takes a variable number of arguments of any type.

```
>>> dbg('choices', range(3))
choices [0, 1, 2]
```

dialogue(*exemplars*={}, *debug*=<function dbg at 0x84b35a4>)

Co-routine for dialogue. Use send(utterance) to receive a reply. Initial input: exemplars Further input: one utterance at a time Output: one reply at a time

edit_dist(*source*, *target*)

Edit distance of two sequences. Non-standard features are that all mutations have a cost of 1 (so as not to favor insertions or deletions over substitutions), as well as a decay based on index (mutations with higher indices weigh less).

```
>>> edit_dist('foo', 'bar')
2.4890507991136213
```

evaluate(*conv*, *exemplars*={}, *n*=1)

When presented with a corpus fragment, feed parent utterances to model and generate model responses in place of child utterances. n = number of iterations to perform Input: string with transcript of corpus Output: same transcript with replies from model instead of child

express(*meaning*, *exemplars*, *lexicon*)

Express 'meaning' by returning the most similar exemplar.

express2(*meaning*, *exemplars*, *constructions*, *lexicon*, *debug*=<function dbg at 0x84b35a4>)

Transform a meaning into a two word utterance using exemplars, constructions or the lexicon. Filter result according to lexical knowledge. Input: meaning representation Output: one or two word utterance

expressmulti(*meaning*, *exemplars*, *constructions*, *lexicon*)

Express 'meaning' by returning a matching exemplar or construction

findmeaning(*form*, *exemplars*, *lexicon*, *debug*=<function dbg at 0x84b35a4>)

Given the words in a construction, find the most common meaning associated with it in the corpus of exemplars.

findtopic(*discourse*, *debug*=<function dbg at 0x84b35a4>)

Look for a recurring element in the last two utterances, if found, this becomes the new topic. Input: discourse (list of utterances and meanings) Output: clause, or None

getexemplars()

Obtain corpus, either by importing from module 'exemplars,' or by falling back to a small sample corpus.

inferconstructions(*exemplars*, *lexicon*, *constructions*={}, *debug*=<function dbg at 0x84b35a4>)

Build corpus of constructions from exemplars and lexicon. Input: exemplars & lexicon. Output: constructions, dictionary of pairings between substrings and clauses.

inferlexicon(*exemplars*, *verbose*=False, *lexicon*={}, *debug*=<function dbg at 0x84b35a4>)

Infer lexicon from corpus of exemplars. Input: exemplars. Output: lexicon, dictionary of word-clause pairings.

interpret(*utterance*, *topic*, *exemplars*, *lexicon*, *debug*=<function dbg at 0x84b35a4>)

Return semantic representation for linguistic utterance. This function is in charge of backtracking over initial exemplars and picking the best result, the rest is done by interpretwith() Input: utterance Output: meaning representation

interpretwith(*words*, *partialmeaning*, *exemplars*, *lexicon*, *debug*=<function dbg at 0x84b35a4>)

Interpretation helper function called by interpret(), work out meaning of remaining words by stitching together the best matching fragments.

intersect(*a*, *b*)

```
>>> intersect([1,2,3], [2,3])
set([2, 3])
```

main()

Interactive textual user interface.

nodbg(**m*)

pred(*token*)

```
>>> pred('foo(bar)')
'foo'
```

reinforce(*meaning*, *reaction*, *reactionutt*, *discourse*, *exemplars*)

Strengthen connection between last two utterances by adding a random identifier and adding or updating the resulting exemplar. Input: two utterances with meanings Side-effect: updated or added exemplars

response(*meaning*, *exemplars*, *debug*=<function dbg at 0x84b35a4>)

Transform a meaning into a response using adjacency pairs of speech acts. Input: meaning representation Output: meaning representation

revdict(*d*)

Reverse dictionary, ie. swap values and keys; since a value may occur with multiple keys, return a list of all keys associated with a value.

```
>>> revdict({0: 1, 1: 2})
{1: [0], 2: [1]}
```

selftalk(*u*, *e*={}, *debug*=<function dbg at 0x84b35a4>)

Talk to oneself, picking a random utterance from the exemplars when repetition is detected. Input: number of utterances to generate, exemplars Output: transcript of conversation

substr_iter(*seq*)

Return all substrings of a sequence, in descending order of length.

```
>>> list(substr_iter('abc'))
['abc', 'ab', 'bc', 'a', 'b', 'c']
```

test(*ex*={}, *lex*={})

Interpret some example utterances.

tokens(*m*)

Turn meaning into a list of operator, predicates and arguments.

```
>>> tokens('whquestion: do(X) animal(bunny)')
['whquestion:', 'do', '(X)', 'animal', '(bunny)']
```

unifies(*meaning*, *partialmeaning*, *debug*=<function dbg at 0x84b35a4>)

succeed if everything in "partialmeaning" is compatible with "meaning", substituting along the way. Similar to conciliate() but operates on the whole meaning instead of looking at individual clauses, and does not perform substitution, merely instantiation.

```
>>> unifies('assertion: do(hop) animal(bunny)',
 ['assertion: do(X) animal(bunny)'])
substituted (X) with (hop)
True
>>> unifies('assertion: do(hop) animal(bunny)',
 ['assertion: animal(bunny) do(X)'])
False
```

var(*arg*)

Test whether an argument string is variable:

```
>>> var('(bar)')
False
>>> var('(X)')
True
```