Update: We've improved the model schematic, and added clarifications to the documentation in both this notebook and in `rnnlm.py`. Fetch the latest version from GitHub, or look at the diffs to see what's new.

Update: We've provided the solution implementation of `MakeFancyRNNCell`; hopefully this will make things easier! For an illustration of how a multi-layer RNN cell works, see this diagram (RNNLM - multicell.png). The "cell" from `MakeFancyRNNCell` with `num_layers = 2` is the unit inside the dashed green box.

Assignment 1, Part 2: RNN Language Model

(45 points total)

In this part of the assignment, you'll implement a recurrent neural network language model. This class of models represents the cutting edge in language modeling, and what you implement will be include many of the same features.

As a reference, you may want to review the following papers:

- Recurrent neural network based language model (http://www.fit.vutbr.cz/research/groups/speech/publi/2010/mikolov_interspeech2010_IS100722.pdf) (Mikolov, et al. 2010)
- Exploring the Limits of Language Modeling (http://arxiv.org/pdf/1602.02410.pdf) (Jozefowicz, et al. 2016)

You'll be writing a fair amount of TensorFlow code, so you may want to review the Week 1 TensorFlow tutorial (../week1/TensorFlow%20Tutorial.ipynb) or the Week 4 notebook (../week4/Neural%20Probabilistic%20Language%20Model.ipynb) for review. For documentation on specific functions, consult the TensorFlow API reference (https://www.tensorflow.org/versions/r0.10/api_docs/python/index.html).

Be sure you're using a recent installation of **TensorFlow version 0.9.0 or higher**.

Note on Indentation

This notebook (as well as rnnlm.py) uses 2-space indentation, instead of the 4 spaces that is Python's default. Why? It makes lines shorter, which is handy if you have a lot of nested scopes. Some people will yell at you for doing this, but the instructors don't care either way.

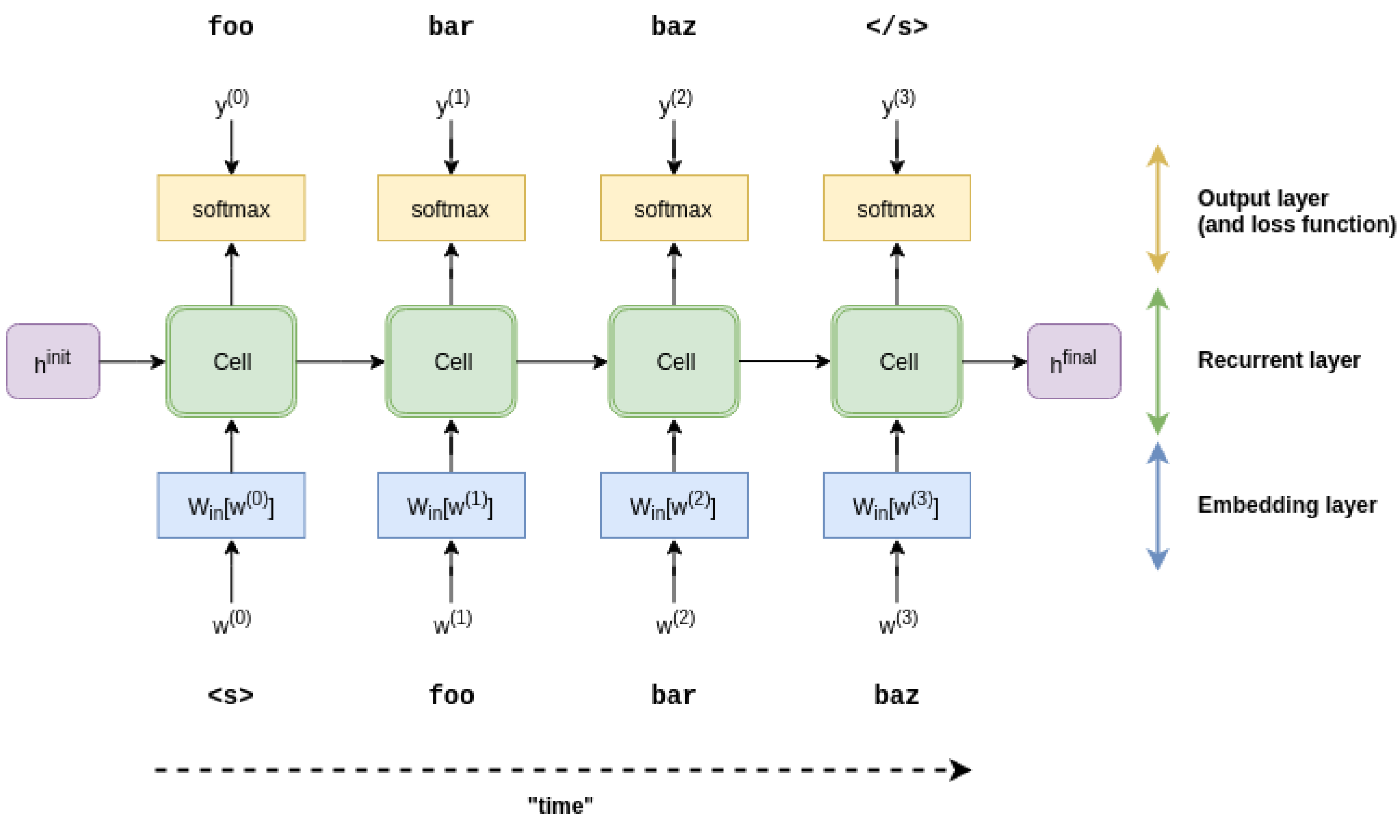
You can follow these instructions (http://jupyter-notebook.readthedocs.io/en/latest/frontend_config.html#example-changing-the-notebook-s-default-indentation) to configure Jupyter to be cool about this. In Chrome, you can open a JavaScript console by going to Menu -> More Tools -> Developer Tools and clicking the "Console" tab.

Part 2 Overview

- **(a)** RNNLM Inputs and Parameters (written questions)
- **(b)** Implementing the RNNLM
- **(c)** Training your RNNLM
- **(d)** Sampling Sentences
- **(e)** Linguistic Properties

In [1]:
```
import os, sys, re, json, time, shutil
import itertools
import collections
from IPython.display import display

# NLTK for NLP utils and corpora
import nltk

# NumPy and TensorFlow
import numpy as np
import tensorflow as tf

# Pandas because pandas are awesome, and for pretty-printing
import pandas as pd
# Set pandas floating point display
pd.set_option('float_format', lambda f: "{0:.04f}".format(f))

# Helper libraries for this notebook
import utils; reload(utils)
import vocabulary; reload(vocabulary)
import rnnlm; reload(rnnlm)
```

RNNLM Model



Here's the basic spec for our model. We'll use the following notation:

- $w^{(i)}$ for the i^{th} word of the sequence (as an integer index)
- $x^{(i)}$ for the vector representation of $w^{(i)}$
- $h^{(i)}$ for the i^{th} hidden state, with indices as in Section 4.8 of the async.
- $o^{(i)}$ for the i^{th} output state, which may or may not be the same as the hidden state

Let $h^{(-1)} = h^{init}$ be an initial state. For an input sequence of n words and $i = 0, \ldots, n-1$, we have:

- **Embedding layer:** $x^{(i)} = W_{in}[w^{(i)}]$
- **Recurrent layer:** $(h^{(i)}, o^{(i)}) = \mathrm{CellFunc}(x^{(i)}, h^{(i-1)})$
- **Output layer:** $\hat{P}(w^{(i+1)}) = \mathrm{softmax}(o^{(i)} W_{out} + b_{out})$

$\mathrm{CellFunc}$ can be an arbitrary function representing our recurrent cell - it can be a simple RNN cell, or something more complicated like an LSTM, or even a stacked multi-layer cell.

We'll use these as shorthand for important dimensions:

- `V` : vocabulary size
- `H` : hidden state size = embedding size

It may be convenient to deal with the logits of the output layer, which are the un-normalized inputs to the softmax:

$$\mathrm{logits}^{(i)} = o^{(i)} W_{out} + b_{out}$$

(a) RNNLM Inputs and Parameters (9 points)

(written - no code) Write your answers in the markdown cell below.

1. Let CellFunc be a simple RNN cell (see Section 4.8). Write the functional form in terms of nonlinearities and matrix multiplication. How many parameters (matrix or vector elements) are there for this cell, in terms of `V` and `H`?
2. How many parameters are in the embedding layer? In the output layer? (By parameters, we mean total number of matrix elements across all train-able tensors.)
3. How many floating point operations are required to compute $\hat{P}(w^{(i+1)})$ for a given target word $w^{(i+1)}$, assuming $h^{(i-1)}$ is already computed? How about for all target words?
4. How does your answer to 3. change if we approximate $\hat{P}(w^{(i+1)})$ with a sampled softmax with k samples? How about if we use a hierarchical softmax? (*Recall that hierarchical softmax makes a series of left/right decisions using a binary classifier $P_s(\text{right}) = \sigma(u_s \cdot o^{(i)} + b_s) \geq 0.5$ at each split s in the tree.*)
5. If you have an LSTM with $H = 200$ and use sampled softmax with $k = 100$, what part of the network takes up the most computation time during training? (*Choose "embedding layer", "recurrent layer", or "output layer"*)

Note: for $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times l}$, computing the matrix product AB takes $O(mnl)$ time.

Answers for Part (a)

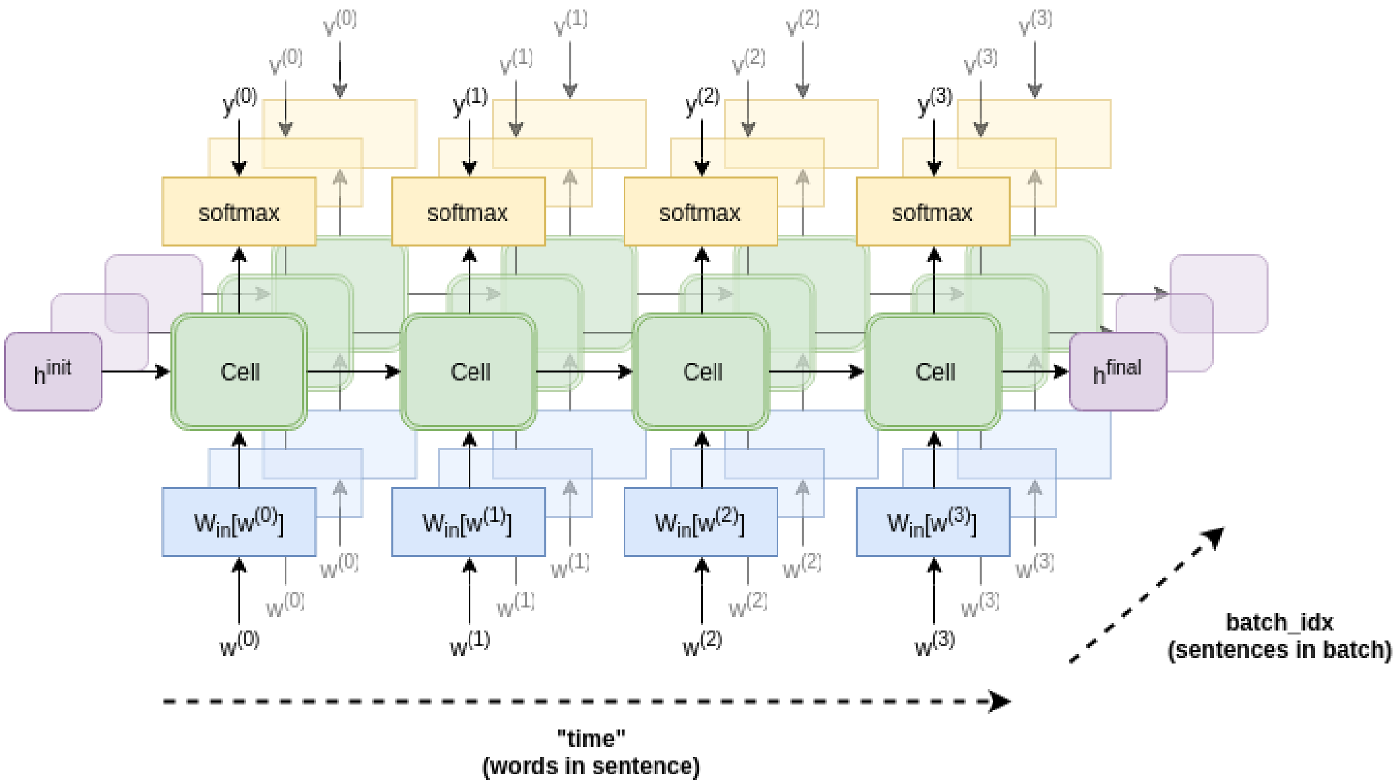
You can use LaTeX to typeset math, e.g. `$ f(x) = x^2 $` will render as $f(x) = x^2$.

1. *Your text here!*
2. *Your text here!*
3. *Your text here!*
4. *Your text here!*
5. *Your text here!*

Batching and Truncated BPTT

In theory, we model our RNN as operating on a single sequence of arbitrary length. During training, we'd run the RNN over that entire sequence, then backpropagate the errors from all the training labels.

In practice, however, it's more common to operate on batches, and to perform *truncated backpropagation* on a fixed-length slice. This means our inputs w and targets y will be 2D arrays of shape `[batch_size, max_time]`:



Batching for an RNN means we'll run several copies of the RNN simultaneously, each with their own hidden state and outputs. Most TensorFlow functions are batch-aware, and expect the batch size as the first dimension.

Truncated backpropagation means that we'll chop our sequences into blocks of fixed length - say, 20 - and run the RNN for only a fixed number `max_time` steps before we backpropagate errors. We'll still keep the final hidden state so that we can keep computing over the sequence, but we won't backpropagate across this boundary. For example, if our input was the sequence `1 2 3 4 5 6 7 8 9 10 11`, and `max_time=5`, we'd do:

- Initialize `h0`
- Run on block `[1 2 3 4 5]`
- Backprop errors from targets `[2 3 4 5 6]`
- Set `h0 = h_final`
- Run on block `[6 7 8 9 10]`
- Backprop errors from targets `[7 8 9 10 11]`

And so on for longer sequences.

(b) Implementing the RNNLM (21 points)

In order to better manage the model parameters, we'll implement our RNNLM in the `RNNLM` class in `rnnlm.py`. We've given you a skeleton of starter code for this, but the bulk of the implementation is left to you.

Particularly, you'll need to implement three functions:

- `BuildCoreGraph()` : the main RNN itself
- `BuildTrainGraph()` : the training operations, including `train_loss_`, and `train_step_`
- `BuildSamplerGraph()` : operations to generate output samples (`pred_samples_`)

See `rnnlm.py` for more documentation.

Notes and Tips

`BuildCoreGraph` We recommend implementing the `MakeFancyRNNCell` function as a wrapper to construct LSTM cells with (optional) dropout and multi-layer cells.

You should use **`tf.nn.dynamic_rnn`** to build your recurrent layer. It takes care of making the recurrent connections and ensuring that the computation is done in the right (temporal) order, and gives you a nice wrapper that can take inputs of shape `[batch_size, max_time, ...]`.

You'll need to provide initializations for your variables in the embedding layer and the output layer; we recommend random uniform or Xavier initialization (as in Part 0). The `tf.nn.rnn_cell` functions will automatically handle initialization of the internal cell variables (i.e. the LSTM matricies).

`BuildTrainGraph`
You can use the softmax loss from `BuildCoreGraph` here, but we strongly recommend implementing an approximate (e.g. sampled) loss function for `train_loss_`. This will greatly speed up training.

For training steps, you can use any optimizer, but we recommend `tf.train.GradientDescentOptimizer` with gradient clipping (`tf.clip_by_global_norm`) or `tf.train.AdagradOptimizer`.

You may find the following API functions useful:

- tf.nn.rnn_cell (https://www.tensorflow.org/versions/r0.11/api_docs/python/rnn_cell.html#RNNCell) (particularly `cell.zero_state`)
- tf.nn.dynamic_rnn (https://www.tensorflow.org/versions/r0.11/api_docs/python/nn.html#dynamic_rnn)
- tf.nn.sparse_softmax_cross_entropy_with_logits (https://www.tensorflow.org/versions/r0.11/api_docs/python/nn.html#sparse_softmax_cross_entropy_with_lo
- tf.nn.sampled_softmax_loss (https://www.tensorflow.org/versions/r0.11/api_docs/python/nn.html#sampled_softmax_loss)
- tf.multinomial (https://www.tensorflow.org/versions/r0.11/api_docs/python/constant_op.html#multinomial)

Additionally, you can expect to make heavy use of tf.shape (https://www.tensorflow.org/versions/r0.11/api_docs/python/array_ops.html#shape) and tf.reshape (https://www.tensorflow.org/versions/r0.11/api_docs/python/array_ops.html#reshape). Note especially that you can use `-1` as a dimension in `tf.reshape` to automatically infer the size. For example:

```
x = tf.zeros([5,10], dtype=tf.float32)
x.reshape([-1,])  # shape [50,]
x.reshape([1, -1])  # shape [1, 50]
```

In [2]:
```
import tensorflow as tf
import rnnlm; reload(rnnlm)

# Clear old log directory
shutil.rmtree("tf_summaries", ignore_errors=True)

with tf.Graph().as_default():
  tf.set_random_seed(42)

  lm = rnnlm.RNNLM(V=10000, H=200, num_layers=2)
  lm.BuildCoreGraph()
  lm.BuildTrainGraph()
  lm.BuildSamplerGraph()

  summary_writer = tf.train.SummaryWriter("tf_summaries",
                                          tf.get_default_graph())
```

The code above will load your implementation, construct the graph, and write a logdir for TensorBoard. You can bring up TensorBoard with:

```
tensorboard --logdir tf_summaries --port 6006
```

As usual, check http://localhost:6006/#graphs (http://localhost:6006/#graphs) to inspect your implementation. Remember, judicious use of `tf.name_scope()` and/or `tf.variable_scope()` will greatly improve the visualization, and make code easier to debug.

(c) Training your RNNLM (5 points)

We'll give you data loader functions in **utils.py**. They work similarly to the loaders in the Week 4 notebook.

Particularly, `utils.batch_generator` will return an iterator that yields minibatches in the correct format. Batches will be of size `[batch_size, max_time]`, and consecutive batches will line up along rows so that the final state h^{final} of one batch can be used as the initial state h^{init} for the next.

For example, using a toy corpus:

In [3]:
```
toy_corpus = "<s> Mary had a little lamb . <s> The lamb was white as snow . <s
>"
toy_corpus = np.array(toy_corpus.split())

print "Input words:"
bi = utils.batch_generator(toy_corpus, batch_size=2, max_time=4)
for i, (w,y) in enumerate(bi):
  utils.pretty_print_matrix(w, cols=["w_%d" % d for d in range(w.shape[1])], d
type=object)

print "Target words:"
bi = utils.batch_generator(toy_corpus, batch_size=2, max_time=4)
for i, (w,y) in enumerate(bi):
  utils.pretty_print_matrix(y, cols=["y_%d" % d for d in range(w.shape[1])], d
type=object)
```

Note that the data we feed to our model will be word indices, but the shape will be the same.

1. Implement the `run_epoch` function

We've given you some starter code for logging progress; fill this in with actual call(s) to `session.run` with the appropriate arguments to run a training step.

Be sure to handle the initial state properly at the beginning of an epoch, and remember to carry over the final state from each batch and use it as the initial state for the next.

Note: we provide a `train=True` flag to enable train mode. If `train=False`, this function can also be used for scoring the dataset - see `score_dataset()` below.

In [4]:
```
def run_epoch(lm, session, batch_iterator, train=False,
              verbose=False, tick_s=10,
              keep_prob=1.0, learning_rate=0.1):
  start_time = time.time()
  tick_time = start_time  # for showing status
  total_cost = 0.0  # total cost, summed over all words
  total_words = 0

  if train:
    train_op = lm.train_step_
    keep_prob = keep_prob
    loss = lm.train_loss_
  else:
    train_op = tf.no_op()
    keep_prob = 1.0  # no dropout at test time
    loss = lm.loss_  # true loss, if train_loss is an approximation

  for i, (w, y) in enumerate(batch_iterator):
    cost = 0.0
    #### YOUR CODE HERE ####

    # At first batch in epoch, get a clean intitial state
    if i == 0:
      h = session.run(lm.initial_h_, {lm.input_w_: w})



    #### END(YOUR CODE) ####
    total_cost += cost
    total_words += w.size  # w.size = batch_size * max_time

    ##
    # Print average loss-so-far for epoch
    # If using train_loss_, this may be an underestimate.
    if verbose and (time.time() - tick_time >= tick_s):
      avg_cost = total_cost / total_words
      avg_wps = total_words / (time.time() - start_time)
      print "[batch %d]: seen %d words at %d wps, loss = %.3f" % (i,
          total_words, avg_wps, avg_cost)
      tick_time = time.time()  # reset time ticker

  return total_cost / total_words
```

2. Run Training

We'll give you the outline of the training procedure, but you'll need to fill in a call to your `run_epoch` function.

At the end of training, we use a `tf.train.Saver` to save a copy of the model to `./tf_saved/rnnlm_trained`. You'll want to load this from disk to work on later parts of the assignment; see **part (d)** for an example of how this is done.

Tuning Hyperparameters

With a sampled softmax loss, the default hyperparameters should train 5 epochs in under 10 minutes on a 16-core Google Cloud Compute instance, and reach a training set perplexity below 200.

However, it's possible to do significantly better. Try experimenting with multiple RNN layers (`num_layers` > 1) or a larger hidden state - though you may also need to adjust the learning rate and number of epochs for a larger model.

You can also experiment with a larger vocabulary. This will look worse for perplexity, but will be a better model overall as it won't treat so many words as `<unk>`.

Submitting your model

You should submit your trained model along with the assignment. Do:

```
git add tf_saved/rnnlm_trained tf_saved/rnnlm_trained.meta
git commit -m "Adding trained model."
```

Unless you train a very large model, these files should be < 50 MB and no problem for git to handle.

In [5]:
```
# Load the dataset
V = 10000
vocab, train_ids, test_ids = utils.load_corpus("brown", split=0.8, V=V, shuffl
e=42)
```

In [6]:
```
# Training parameters
max_time = 20
batch_size = 50
learning_rate = 0.5
keep_prob = 1.0
num_epochs = 5

# Model parameters
model_params = dict(V=V,
                    H=100,
                    num_layers=1)

trained_filename = 'tf_saved/rnnlm_trained'
```

In [7]:
```
def score_dataset(lm, session, ids, name="Data"):
  bi = utils.batch_generator(ids, batch_size=100, max_time=100)
  cost = run_epoch(lm, session, bi,
                   learning_rate=1.0, keep_prob=1.0,
                   train=False, verbose=False, tick_s=3600)
  print "%s: avg. loss: %.03f  (perplexity: %.02f)" % (name, cost,
np.exp(cost))
```

In [8]:
```
# Will print status every this many seconds
print_interval = 5

# Clear old log directory
shutil.rmtree("tf_summaries", ignore_errors=True)

with tf.Graph().as_default(), tf.Session() as session:
  # Seed RNG for repeatability
  tf.set_random_seed(42)

  with tf.variable_scope("model", reuse=None):
    lm = rnnlm.RNNLM(**model_params)
    lm.BuildCoreGraph()
    lm.BuildTrainGraph()

  session.run(tf.initialize_all_variables())
  saver = tf.train.Saver()

  for epoch in xrange(1,num_epochs+1):
    t0_epoch = time.time()
    bi = utils.batch_generator(train_ids, batch_size, max_time)
    print "[epoch %d] Starting epoch %d" % (epoch, epoch)
    #### YOUR CODE HERE ####

    # Run a training epoch.

    #### END(YOUR CODE) ####
    print "[epoch %d] Completed in %s" % (epoch,
utils.pretty_timedelta(since=t0_epoch))

    ##
    # score_dataset will run a forward pass over the entire dataset
    # and report perplexity scores. This can be slow (around 1/2 to
    # 1/4 as long as a full epoch), so you may want to comment it out
    # to speed up training on a slow machine. Be sure to run it at the
    # end to evaluate your score.
    print ("[epoch %d]" % epoch),
    score_dataset(lm, session, train_ids, name="Train set")
    print ("[epoch %d]" % epoch),
    score_dataset(lm, session, test_ids, name="Test set")
    print ""

    # Save a checkpoint
    saver.save(session, 'tf_saved/rnnlm', global_step=epoch)

  # Save final model
  saver.save(session, trained_filename)
```

(d) Sampling Sentences (5 points)

If you didn't already in **part (b)**, implement the `BuildSamplerGraph()` method in `rnnlm.py` See the function docstring for more information.

Implement the `sample_step()` method below (5 points)

This should access the Tensors you create in `BuildSamplerGraph()`. Given an input batch and initial states, it should return a vector of shape `[batch_size,]` containing sampled indices for the next word of each batch sequence.

Run the method using the provided code to generate 10 sentences.

In [9]:
```
def sample_step(lm, session, input_w, initial_h):
  """Run a single RNN step and return sampled predictions.

  Args:
    lm : rnnlm.RNNLM
    session: tf.Session
    input_w : [batch_size] list of indices
    initial_h : [batch_size, hidden_dims]

  Returns:
    final_h : final hidden state, compatible with initial_h
    samples : [batch_size, 1] vector of indices
  """
  #### YOUR CODE HERE ####
  # Reshape input to column vector
  input_w = np.array(input_w, dtype=np.int32).reshape([-1,1])

  # Run sample ops

  #### END(YOUR CODE) ####
  return final_h, samples[:,-1,:]
```

In [10]:
```
# Same as above, but as a batch
max_steps = 20
num_samples = 10
random_seed = 42

with tf.Graph().as_default(), tf.Session() as session:
  # Seed RNG for repeatability
  tf.set_random_seed(random_seed)

  with tf.variable_scope("model", reuse=None):
    lm = rnnlm.RNNLM(**model_params)
    lm.BuildCoreGraph()
    lm.BuildSamplerGraph()

  # Load the trained model
  saver = tf.train.Saver()
  saver.restore(session, trained_filename)

  # Make initial state for a batch with batch_size = num_samples
  w = np.repeat([[vocab.START_ID]], num_samples, axis=0)
  h = session.run(lm.initial_h_, {lm.input_w_: w})
  # We'll take one step for each sequence on each iteration
  for i in xrange(max_steps):
    state, y = sample_step(lm, session, w[:,-1:], h)
    w = np.hstack((w,y))

  # Print generated sentences
  for row in w:
    for i, word_id in enumerate(row):
      print vocab.id_to_word[word_id],
        if (i != 0) and (word_id == vocab.START_ID):
          break
    print ""
```

(e) Linguistic Properties (5 points)

Now that we've trained our RNNLM, let's test a few properties of the model to see how well it learns linguistic phenomena. We'll do this with a scoring task: given two or more test sentences, our model should score the more plausible (or more correct) sentence with a higher log-probability.

We'll define a scoring function to help us:

In [11]:
```
def score_seq(lm, session, seq, vocab):
  """Score a sequence of words. Returns total log-probability."""
  padded_ids = vocab.words_to_ids(utils.canonicalize_words(["<s>"] + seq,
                                                            wordset=vocab.word_
to_id))
  w = np.reshape(padded_ids[:-1], [1,-1])
  y = np.reshape(padded_ids[1:],  [1,-1])
  h = session.run(lm.initial_h_, {lm.input_w_: w})
  feed_dict = {lm.input_w_:w,
               lm.target_y_:y,
               lm.initial_h_:h,
               lm.dropout_keep_prob_: 1.0}
  # Return log(P(seq)) = -1*loss
  return -1*session.run(lm.loss_, feed_dict)

def load_and_score(inputs, sort=False):
  """Load the trained model and score the given words."""
  with tf.Graph().as_default(), tf.Session() as session:
    with tf.variable_scope("model", reuse=None):
      lm = rnnlm.RNNLM(**model_params)
      lm.BuildCoreGraph()

    # Load the trained model
    saver = tf.train.Saver()
    saver.restore(session, trained_filename)

    if isinstance(inputs[0], str) or isinstance(inputs[0], unicode):
      inputs = [inputs]

    # Actually run scoring
    results = []
    for words in inputs:
      score = score_seq(lm, session, words, vocab)
      results.append((score, words))

    # Sort if requested
    if sort: results = sorted(results, reverse=True)

    # Print results
    for score, words in results:
      print "\"%s\" : %.02f" % (" ".join(words), score)
```

Now we can test as:

In [12]:
```
sents = ["once upon a time",
         "the quick brown fox jumps over the lazy dog"]
load_and_score([s.split() for s in sents])
```

1. Number agreement

Compare **"the boy and the girl [are/is]"**. Which is more plausible according to your model?

If your model doesn't order them correctly (*this is OK*), why do you think that might be? (answer in cell below)

In [13]:
```
#### YOUR CODE HERE ####


#### END(YOUR CODE) ####
```

Answer to part 1. question(s)

Answer to above question(s).

2. Type/semantic agreement

Compare:

- **"peanuts are my favorite kind of [nut/vegetable]"**
- **"when I'm hungry I really prefer to [eat/drink]"**

Of each pair, which is more plausible according to your model?

How would you expect a 3-gram language model to perform at this example? How about a 5-gram model? (answer in cell below)

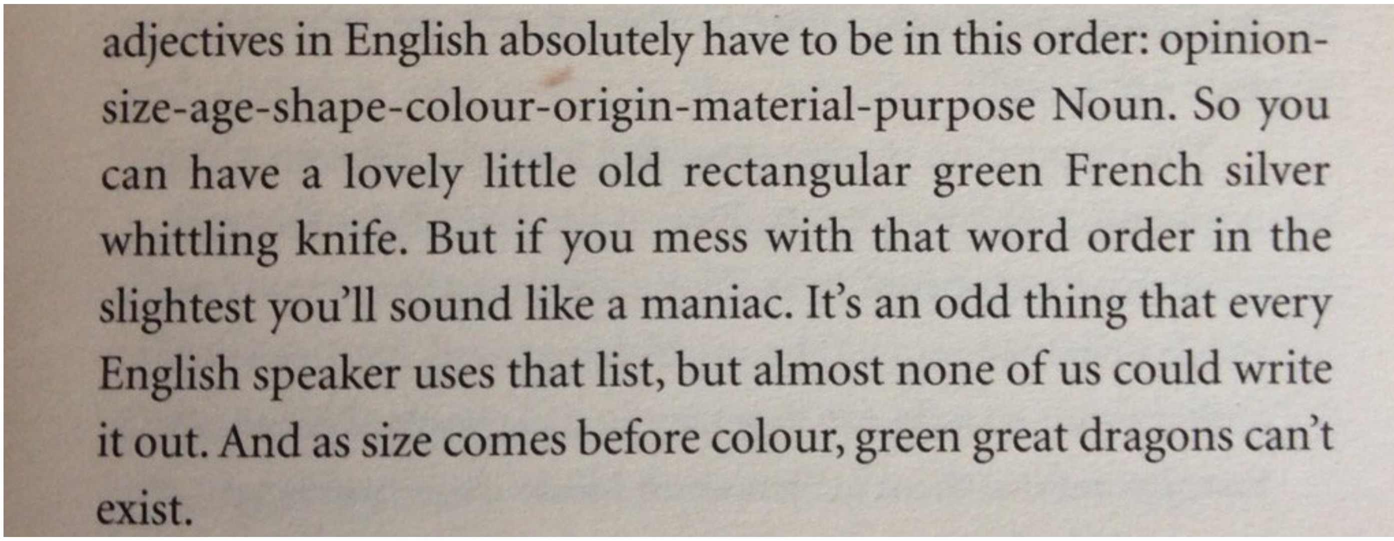
In [14]:
```
#### YOUR CODE HERE ####



#### END(YOUR CODE) ####
```

Answer to part 2. question(s)

Answer to above question(s).

3. Adjective ordering (just for fun)

Let's repeat the exercise from Week 2:


adjectives in English absolutely have to be in this order: opinion-size-age-shape-colour-origin-material-purpose Noun. So you can have a lovely little old rectangular green French silver whittling knife. But if you mess with that word order in the slightest you'll sound like a maniac. It's an odd thing that every English speaker uses that list, but almost none of us could write it out. And as size comes before colour, green great dragons can't exist.

source: [https://twitter.com/MattAndersonBBC/status/772002757222002688?lang=en (https://twitter.com/MattAndersonBBC/status/772002757222002688?lang=en)](https://twitter.com/MattAndersonBBC/status/772002757222002688?lang=en)

We'll consider a toy example (literally), and consider all possible adjective permutations.

Note that this is somewhat sensitive to training, and even a good language model might not get it all correct. Why might the NN fail, if the trigram model from week2 was able to solve it?

In [15]:
```
prefix = "I have lots of".split()
noun = "toys"
adjectives = ["square", "green", "plastic"]
inputs = []
for adjs in itertools.permutations(adjectives):
  words = prefix + list(adjs) + [noun]
  inputs.append(words)

load_and_score(inputs, sort=True)
```

In [ ]:
```
```