CMPT-310 Homework One

Andrew Poelstra

October 7, 2010

1 Problem Statement

Our goal is, given a sentence that conforms to a specific grammar (a subset of the English language), to convert it from the active voice to the passive voice. This often aids readability.

2 Strategy

2.1 Overall Strategy

Because sentences are listed linearly, but interpreted as trees, our first step will be to convert the input from a linked-list format to a parse tree. On the way, we will tag each subtree and word with a symbol denoting its grammatic purpose. This will let the computer manipulate the sentence without worrying about the specific words involved.

Using that parse tree, we will re-arrange the sentence. We will traverse the tree, looking for any `:RC` nodes, whose children we will rearrange. (Specifically, we will convert `:RC` nodes of the active form `:THAT :NP :V` to the passive form `:THAT :ARE: :V-PAST :BY :NP`.)

Everything else will be left alone.

Finally, we will flatten the transformed parse tree back into a sentence for human consumption.

2.2 Detailed Strategy

2.2.1 Storage Structures

The input sentences will be given in the form of simple Lisp lists of symbols. We will translate these lists into a tree-list structure. Each node will be defined as a list:

$$n_i = (s_1, s_2, \ldots s_n)$$

with each sub-node t_i defined as:

$$s_i = \texttt{(:SYM . sub-node)}$$

wherein `:SYM` is a non-terminal symbol from the grammar definition, and `sub-node` is either a list of sub-nodes, or a terminal symbol from the sentence, such as `'dogs`.

For example, the sentence *dogs eat* would be translated to the following tree, in Lisp syntax:

```
(( :S . ((:NP . ((:N . dogs))) (:VP . ((:V . eat))) ))
```

For simplicity's sake, we will avoid mixing terminal and non-terminal children of any node. To this end, we will introduce "non-terminals" such as `:THAT` and `:ARE`, which may only correspond to one word (themselves).

In addition to the parse tree, we store valid terminals in Lisp lists such as `*nouns*` and `*verbs*`.

2.2.2 Algorithms

Parse Tree Generation The first step is to convert a sentence into a parse tree. We do so recursively, by the following procedure:

1. Check sentence for remaining words.
   (a) If there are none, indicating failure, or exactly one — `:END-TOKEN` — we are done, so we stop here and return the current parse tree.
   (b) Otherwise, there are words to parse, so we loop on the current tree to find legal potential parses. Note that because of our decision not to mix terminals and non-terminals, we know that all sub-nodes will be non-terminals. (If they were terminals, the parse would be done, so we wouldn't have called the function in the first place!)
      i. If the node can be replaced by a series of non-terminals, we do that. If there are multiple choices, we try the longest one first to avoid false positives (which would pop words off the sentence and then bomb later). If one succeeds, excellent! Otherwise, we have failed, so we set the sentence to `NIL`.
      ii. Whenever we insert a new sequence of non-terminals, we recurse on that new sequence to attempt to complete the tree.
      iii. If the node can be replaced by a terminal, we check the sentence to make sure that the first unparsed word matches. If it does, we pop the word off the sentence and into the parse tree. If not, we have failed, so we set the sentence to `NIL`.
2. After the loop, we return the remainder of the sentence (which may be `NIL` if we failed the parse) and the current parse tree. At the top level of recursion, the remainder will consist of either `:END-TOKEN` or `NIL`, and the current parse tree will be the complete parse tree.

Parse Tree Transformation To convert the parse tree from active voice to passive voice, we use the following procedure, which destructively modifies the tree:

1. Loop through the current node, unless it is a terminal.
2. If the current sub-node is `:RC`, re-arrange its children:
   (a) Replace the structure `:THAT :NP :V` with `:THAT :ARE: :V-PAST :BY :NP`.
   (b) The sub-node `:V` is converted to the corresponding past tense `:V-PAST`.
   (c) The sub-node `:NP` is recursed on, since it may contain additional `:RC` nodes.
3. Otherwise, recurse on its children.

Sentence Conversion Finally, we convert the parse tree back into a sentence, by the following procedure:

1. Set *sentence* to `NIL`.
2. Loop through the current node, looking at its child.
   (a) If the child is a terminal, append it to *sentence*.
   (b) Else, it is a list of sub-nodes, so recurse on it.

Final Result The final function, `(convert (sentence))`, works as follows:

1. Add `:END-TOKEN` to the sentence as a sentinel.
2. Convert the sentence to a parse tree.
   (a) If the returned sentence is entirely `:END-TOKEN`, excellent!
   (b) Otherwise, the parse failed, so we return `:ERRONEOUS`.
3. Transform the parse tree from active to passive voice.
4. Convert the parse tree back to a sentence, and return it.

2.3 Testing Strategy

There were three major areas of testing:

2.3.1 Simple Tests

First I ran the converter using simple, correct sentences that could be easily verified by hand. (For example, *dogs that frogs see chase cats.*) During this phase, I outputted the parse tree before and after transformation to ensure that each word was being interpreted correctly.

2.3.2 Complex Testing

Then, I tested the parser with very long sentences and sentences with internal recursion, such as *dogs that frogs that cats that dogs avoid see hit bugs that frogs that dogs avoid love.* These sentences were valid, but not as easy to verify. The parser handled them correctly.

2.3.3 Invalid Testing

Finally, I tested the parser with invalid input. First, I added words that the parser did not understand. As expected, it returned these words unparsed, before `:END-TOKEN`, and the conversion function marked it as erroneous.

Then, I created sentences that contained valid words but were not grammatical. (For example, *dogs cats see chase.*) As expected, the returned sentence was simply `NIL`, and the conversion function marked it erroneous.

Finally, I combined the techniques. There were no false positives.

3 Attached

Attached to this document are three things:

1. A session log of part 1 (writing and testing simple functions)
2. A session log of part 2 (writing and testing the sentence converter)
3. A printout of the Lisp code for the sentence converter