Jess Karol
Assignment 10
Ling 40 - Semantics
Nathan Sanders
Fall 2015

Computational Semantics: Implementation of Simple Montague Translation

A core motivation for Montague semantics is the ability to algorithmically develop a formula that represents the meaning of a sentence through combining the meaning of its parts. In class, we developed this algorithm by defining several operations that each node of a syntax tree could potentially undergo. If we use these operations correctly, we can fill up the nodes of a syntax tree starting at the leaves and ending with a formula that represents the complete sentence at the root. The node operations we first encountered are: non-branching node, functional application, and predicate modification. These functions operate on the lexical translations of words which are found in a lexical dictionary. From the dictionary, our three node operations can combine objects and predicates in various ways depending on their position in the syntax tree. With these tools and a systematic approach for traversing a syntax tree, we can derive sentence meaning without the need for human cleverness. In other words, we can program a computer to do it, which is what I have done for this project.

The first step in the process of translating a sentence is parsing the words into a corresponding syntax tree structure. This process is completed algorithmically; however, instead of implementing my own tool, I used the python natural language toolkit recursive descent parser. The recursive descent algorithm is rather simple, but effective for most simple sentence structures. It attempts to verify the input stream is syntactically correct as it is read from left to right. It uses a model, which I defined in a file named “mygrammer.cfg”, and attempts to fit the

input stream into the model using knowledge of each word's part of speech. The document type ".cfg" stands for Context-Free Grammar because the model is independent of any context in which the sentence is spoken. While this method has flaws for complicated sentence structures and is limited to the completeness of a model, it worked well for this project because the complexity of sentences that my Montague translator can handle is also limited.

After attaining a syntax tree, the next step in deriving a function that represents the sentence's meaning is simply replacing the leaves of the tree (the terminal nodes which contain the words used in the original sentence) with their corresponding lexical translations. I preloaded the lexical translations from a text file into a hashtable data structure for convenience. This allows me to very quickly match up an english word with its translation. My function "baseNodes()" iterates through the leaves of the syntax tree and substitutes each english word for its lexical translation if available. After a tree is structured with lexical entries, we can start applying the Montague translation functions, starting at the leaves, and working upwards.

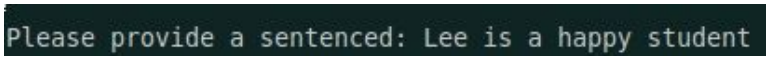
The function that does the most heavy lifting is named "traverse()", which iterates through each level of the tree and applies one of three translation functions. The None-Branching Node function, NN(), simply moves the value of the current node one level up on the tree because the parent of the current node does not branch. The Functional Application function, FA(), combines the predicate or object of the current node with the predicate or object of the neighboring node that has the same parent. This function must insure the predicate and object are in the correct order before combining. Lastly, the Predicate Modification function, PM(), is only activated when the parent node has the same part of speech as one of its children. In this case, a

conjunctive “and” is placed between both predicates. These three operations are called within a loop in “traverse()”, which iterates through each subtree, starting at the bottom and working up.

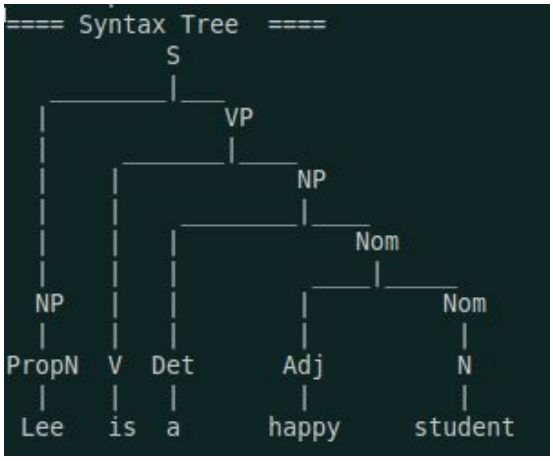
Each time a function is called, the program pauses for 2 seconds to let the user process what is happening, and prints out what operation is being completed as well as the state of the tree. Once the program is finished, the final tree should have the correct translation at the root.

To better understand how the program functions, below are screenshots as well as explanation for the program running through the sentence *Lee is a happy student*.

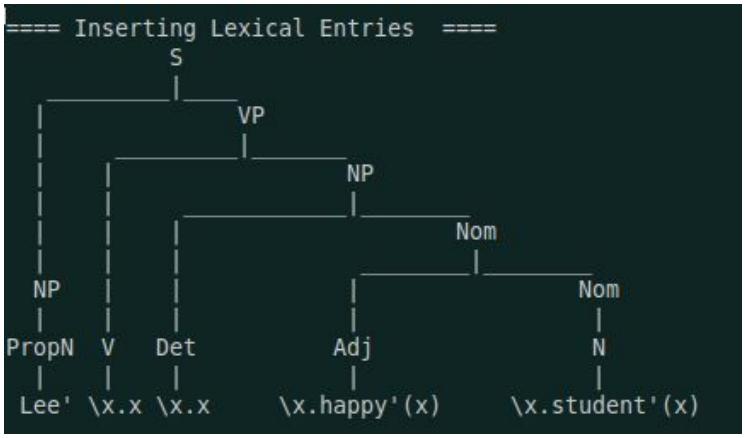
First the user is prompted to enter a sentence:



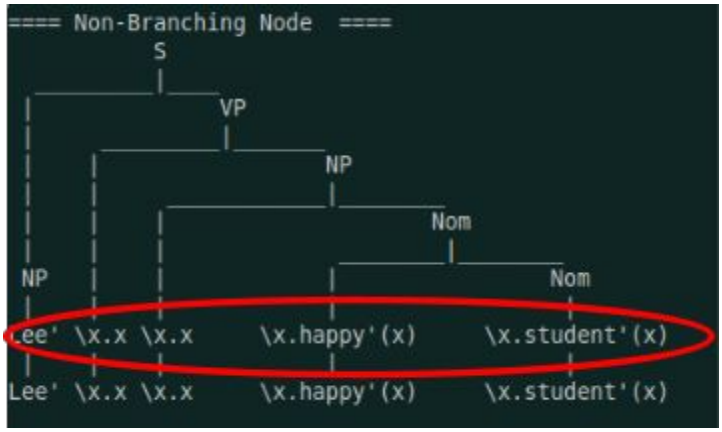
Then, using the recursive descent parser, the program produces a syntax tree for the input:



Next the program replaces each leaf node with the lexical translation. The backslash character is used to represent the usual lambda character.

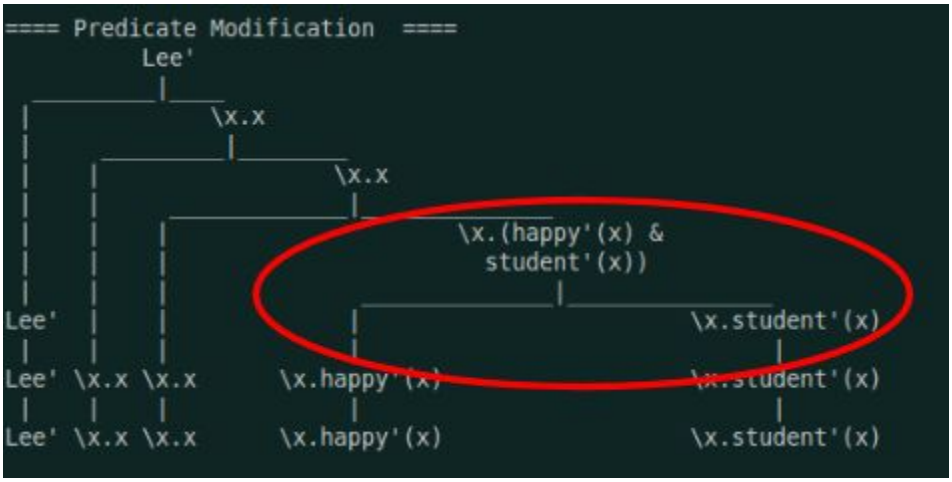


Next, we begin bubbling up the tree to combine nodes. The next step in the process is to apply the Non-Branching Node function because each leaf has a non-branching parent. After the first Non-Branching Node step, the program no longer prints Non-Branching modifications because they are trivial. In the following screen captures, I have placed red circles to better visualize what is changing between steps.

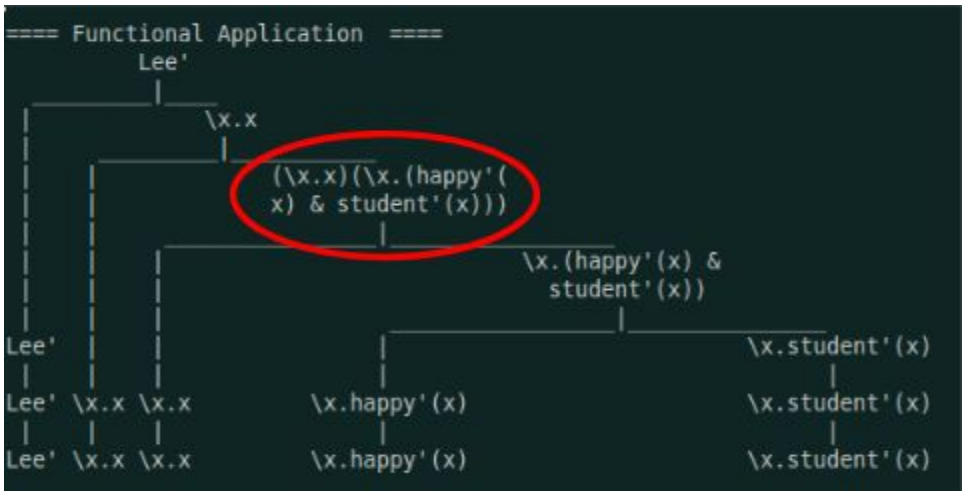


The next step is predicate modification. The algorithm noticed that the parent node and one of the child nodes had the same syntactic label, namely *Nom*, so predicate modification was the appropriate function to apply when combining the children nodes. While the program is

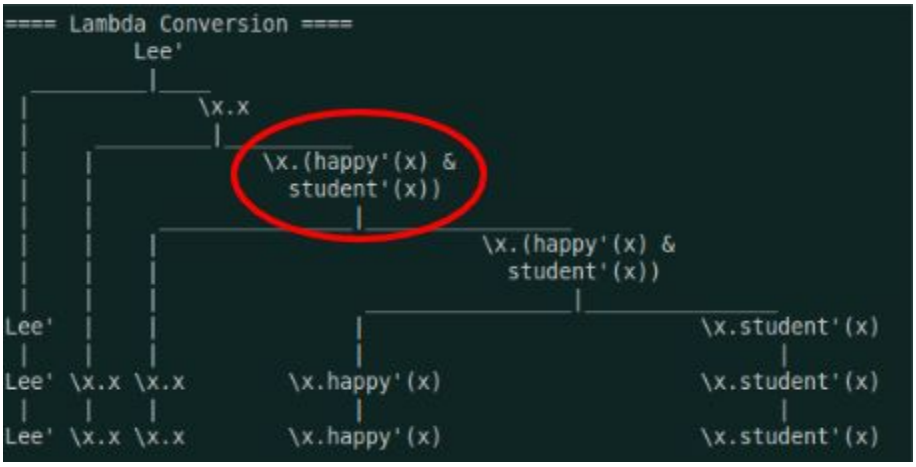
modifying the internal nodes of the tree by replacing the parts of speech with lexical translations, it keeps a replica of the original to check for cases like Predicate Modification.



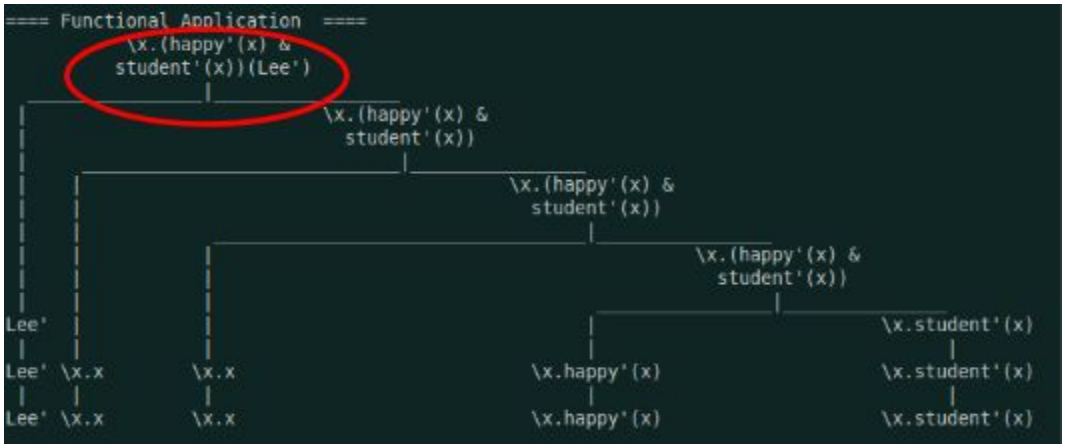
The next step is to combine the translation of *happy student* using functional application with the determiner *a*. Since the *a* is represented by \x.x, which is the identity function, the result should simply be the translation of *happy student*.



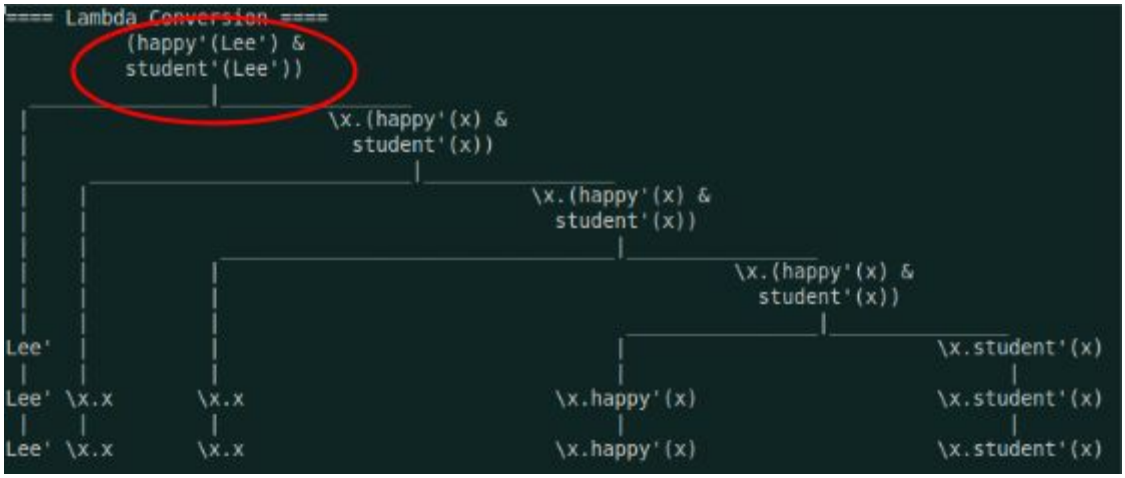
The first step in combining these nodes is to put them together. Then, the expression can be simplified through a process called lambda conversion. Which strips of the first lambda x and plugs in the corresponding argument everywhere in the predicate. The result is as follows:



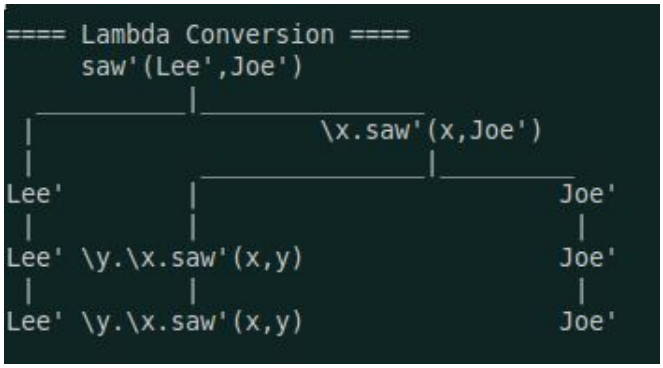
The process of moving up the tree and applying functional application and then lambda conversion continues until we obtain our final result. Below are the final two steps:



The final step is simplifying this expression to arrive at a function that represents the original sentence.



As you can see, the final translation of the sentence is *happy'(Lee') & student'(Lee')*, which is correct. The algorithm not only works for unary predicates, but also works for binary predicates. The following is the last step in translating the sentence *Lee saw Joe.* In this case, *saw* is a binary predicate that takes two arguments, x and y.



My program also provides some error checking for valid english sentences. However, it is often hard to detect whether the syntax parser fails due to incompleteness of the model or if the input stream is invalid. The follow example shows what happens when a non-valid english sentence is the input.

```
Please provide a sentenced: Lee is log
==== Syntax Tree  ====
Syntax tree could not be created. Please check that grammar file contains sentence
structure or that your input is a valid english sentence.
```

The input sentence is *Lee is log*, which is invalid because a determiner is required before the noun. Since the parser could not fit the sentence to a model, an error message is outputted to the user before exiting which explains possible reasons for the parser failing.

While human semantic processing occurs in a complex neurological circuit that is hardly understood by scientists, Montague semantics has attempted to transform our formulation of the meaning of a sentence into something that can be modeled computationally. Through building a very simplified program to step by step bring together the pieces of a sentence into one representative formula, I have come to better understand the process of Montague semantics and the importance it has on understanding sentence meaning in natural language processing. My program does not deal with quantifiers; however, computational semantics could be used to resolve scope ambiguity in constructing meaningful representations. There are many other use cases that are not considered in my project, but using the foundation that I have built, more complexity can be added to account for a wider variety of sentence structures and sentence meanings. This project has lead me to an understanding of one aspect of natural language processing, and I look forward to continuing research and further development of this application using concepts that we learned in Linguistics 040.