A Primer on Tree Automata Software for Natural Language Processing

Jonathan May and Kevin Knight

August 27, 2008

1 Preliminaries

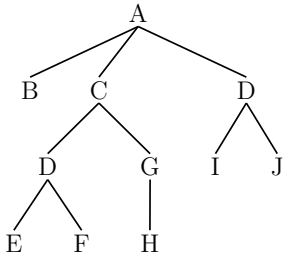
Thank you for reading this document! It is our hope that after reading it you will understand what tree automata are, why you would want to use them for natural language processing applications, and how to use **Tiburon**, the tree automata manipulation software to do so. Although this document attempts to be self-contained, you might feel more comfortable if you have at least a passing knowledge of context-free grammars and/or finite-state (string) automata. For the latter, we recommend the tutorial by Kevin Knight and Yaser Al-Onaizan, which can be freely downloaded from `http://www.isi.edu/licensed-sw/carmel/carmel-tutorial2.pdf`. That document is also a source of inspiration for this one and thus the same concepts are discussed in both. However, the acceptors and transducers (collectively, the automata) described in that document operate on *strings*, while the automata discussed here operate on *trees*. Thus, this document serves as an extension of the previously discussed concepts from the string domain to the tree domain. This is meant to be an informal discussion and not rigorous theory – see the references section at the end of this document for more formal treatments of the theoretical concepts discussed.

2 Strings vs. Trees

Although the basic unit of data we will deal with is the tree, strings are also used in tree automata discussions, as we will see later. A *string*, also known

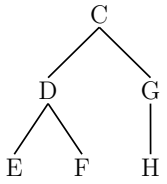
as a *sentence*, is an ordered sequence of 0 or more *symbols* (sometimes known as *words*), drawn from a finite vocabulary. An example of a string is "the boy ran away". A *tree* is somewhat more complicated – it is a symbol drawn from a finite vocabulary and an ordered sequence of 0 or more trees; these trees are called the *subtrees* of the tree. A *leaf* is a tree with no subtrees. The *yield* is a string formed from a tree as follows: If the tree is a leaf, the yield is the leaf (taken as a string); otherwise the yield is the concatenation of the yields of the subtrees.

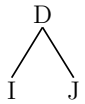
Here is a tree:



And here are the subtrees of that tree:

B



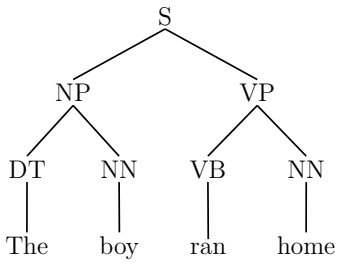


The yield of that tree is the string "B E F H I J".

Because we don't customarily use schematic diagrams in computer text files, we need a convention for representing trees in a text format. By convention, the subtrees of a tree are listed inside a set of parentheses. If a tree has no subtrees, the parentheses are omitted. Thus, the tree above is represented as follows:

```
A(B C(D(E F) G(H)) D(I J))
```

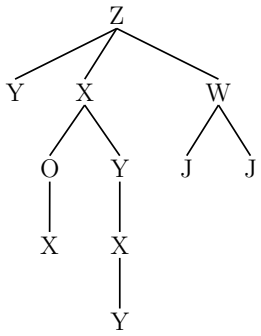
Trees are often used in natural language contexts to represent grammatical structure. Here is a typical syntax parse tree over a sentence:



The yield of this tree is "The boy ran home".

Exercises

- Write this tree in text format:



- Write this tree in schematic format:
  t(l i d(r(n j(g)) u z(i)) g(s) t j(p(i) z(g(c))) a(s))
  What is its yield?

3 Regular Tree Grammars

The tree languages we will work with are the *regular tree languages*, the tree analogue to regular (string) languages. We represent tree languages with a regular tree grammar (RTG). A regular tree grammar is very similar in appearance to the more familiar context-free grammars (CFGs), but has a notion of states. Consider the following simple CFG for noun phrases:

```
NP
NP -> DT NN
```

```
NP -> NN
DT -> the
NN -> boy
NN -> girl
```

CFGs are typically used as a way of representing a class of string languages (known as *context-free languages*). If you're familiar with CFGs, you should skip this very quick tutorial and go on to the section headed "CFGs as Tree Grammars".

A Very Quick Tutorial on CFGs

We already mentioned the example above is called a CFG, and it represents a language of strings. But what do those symbols mean, and how can you figure out the strings represented by the grammar? Briefly put, the CFG consists of a single symbol, called the *start symbol*, and several *rewrite rules*. In the example above, the solo `NP` at the top is the start symbol, and the other five lines with arrows are the rules. Each rule has one symbol to the left of the arrow (called the left side) and one or more symbols to the right (called the right side).

To identify a string in this language, take a piece of paper, and a pencil with an eraser (the eraser is important), and write down the start symbol, Your paper should look like:

```
NP
```

Now, look through the rules in the CFG. Find a rule that has the same symbol to the left of the arrow that you have written on your paper. If there is more than one rule that matches, (as there is in this case) just pick any one, it doesn't matter. Now, *erase* the symbol you have written down and in its place, write whatever is on the right side of the rule you have chosen. So, if you picked the rule `NP -> DT NN`, your paper would now look like:

```
DT NN
```

Now, we're going to keep on erasing and rewriting until we can't do it any more. There's only one rule that matches `DT`, so after you erase and write, your paper would look like:

```
the NN
```

There's no rule that has `the` on its left side, so leave that alone. Find a rule for `NN`. If you made the same decision as we did, your paper would now look like:

```
the boy
```

Since there's no more erasing we can do, we stop. The string we've written down, "`the boy`" is a string in the language represented by the CFG. To get the entire language, just repeat the whole process again, always starting from the start symbol, but make different choices (i.e. you could have chosen `NP -> NN` instead of `NP -> DT NN` or `NN -> boy` instead of `NN -> girl`). The language of this CFG is {`the boy`, `the girl`, `boy`, `girl`}. You should try it yourself to make sure you get the same set.

CFGs as Tree Gramamrs

Although we typically think of CFGs as representing string languages, we can also think of them as representing tree languages, if for every string in the language we also maintain the derivation tree used to form the string. For those of you that read the CFG tutorial, this means instead of erasing a symbol and writing the right side of the rule in its place, you write the right side of the rule *under* the symbol, and draw connecting lines. (For those of you who didn't do the tutorial, bear with us for a second. Or you might want to go back and read it.) Redoing the example we worked through for derivation trees, we still start with:

```
NP
```

(It's probably a good idea to leave a little drawing space on the left this time). Now, instead of erasing, we write the right side of our chosen rule under what we have already written (again, leaving some space)...

```
NP


DT    NN
```

... and then fill in connecting lines.

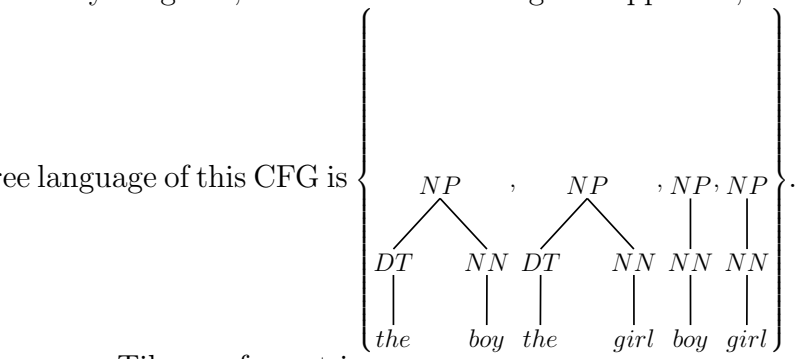
```
    NP
    /\
   /  \
  DT   NN
```

We next handle the DT...

```
    NP
    /\
   /  \
  DT   NN
  |
  |
 the
```

... then the NN...

```
    NP
    /\
   /  \
  DT   NN
   |    |
   |    |
  the  boy
```

... and since we can't add anything else, we're done! Following this approach, you can see that the tree language of this CFG is  .

An analogous RTG in proper Tiburon format is:

```
%% Filename: rtg1 %%
q
```

```
q -> NP(dt nn)
q -> NP(nn)
dt -> DT(dtword)
nn -> NN(nnword)
dtword -> the
nnword -> boy
nnword -> girl
```

An RTG is a lot like a CFG. It has start symbols, which we now call *states*, and it has rules. The rules in an RTG have a single state on the left, and a tree on the right. Like the "string" way of operating a CFG, we start with the start state, then repeatedly erase states and in their place write down the right sides of the rules we used (which now are trees), until there are no states left to replace. Let's try operating the RTG one more time. First, we start by writing down the start state:

```
q
```

Now we find a rule with a left side that matches and erase `q`, replacing it with the left side of that rule:

```
NP(dt nn)
```

(You'll notice that this time we're using the "text format" described above to write trees, but everything should otherwise be fairly familiar). Now we replace `dt` as before (it probably would have been a good idea to leave a little more space)...

```
NP(DT(dtword) nn)
```

We next replace `dtword`.

```
NP(DT(the) nn)
```

Now we replace `nn` (there was no need to replace `dtword` before `nn` – it was an arbitrary choice) and we're almost done.

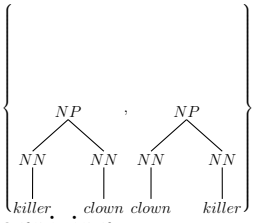
```
NP(DT(the) NN(nnword))
```

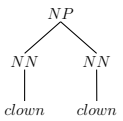
Finally we replace `nnword` and the tree is complete.

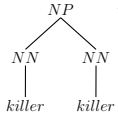
```
NP(DT(the) NN(boy))
```

Pretty simple! In the rest of this document we're going to (mostly) stay away from CFGs and focus more of our attention on RTGs. However, Tiburon supports CFGs and many of the operations you perform on RTGs can also be performed on CFGs. The notion of "state" that is present in RTGs allows us to have more control over the trees we produce than is possible in the "derivation mode" of CFGs. To see why, consider this example:

```
%% Filename: rtg2 %%
q
q -> NP(x1 x2)
q -> NP(x2 x1)
x1 -> NN(a)
x2 -> NN(b)
a -> killer
b -> clown
```

This RTG represents the language 

, but no CFG can recognize these trees without also recognizing 

or 

. Try it for yourself.

You may ask, "Why would I want to use this funny notation to represent a set of trees when I could just write the trees down?" Usually, the trees can be represented much more compactly by writing the RTG than by listing the trees. For example, consider `rtg3`, below:

```
%% filename: rtg3 %%
q
q -> NP(jj nn nn)
jj -> silly
jj -> wild
jj -> green
nn -> monkey
nn -> police
nn -> car
```

That RTG has 27 different trees in it but only took 8 lines to write. Even better, RTGs can represent an *infinite* number of trees:

```
%% filename: rtg4
q
q -> S(np vp)
np -> NP(dt nn)
np -> NP(dt nn pp)
pp -> PP(prep np)
vp -> VP(vb do)
vb -> ran
do -> home
prep -> with
prep -> on
dt -> the
nn -> boy
nn -> monkey
nn -> clown
```

If you wanted to represent this language by listing all the trees in it, you would have a big problem!

We've talked a lot about trees and grammars. Before we go any further, let's take a look at Tiburon, the software which will make it much easier to work with trees and grammars, especially when they get very large. If you're reading this document you probably have also seen instructions for obtaining and installing Tiburon. Follow those instructions and make sure the software works for you. To test this, make sure you are in the directory where tiburon is installed and type this at the command line:

```
% tiburon
```

(For the rest of this document commands you should type will be in typewriter font and next to a "%" sign, indicating the prompt.) You should see something like this:

```
This is Tiburon, version 0.5.0
Error: Parameter 'infiles' is required.
Usage: tiburon
             [-h|--help] (-e|--encoding) <encoding>
     (-m|--semiring) <srtype> [--leftapply]
```

```
[--rightapply] [-b|--batch] [(-a|--align) <align>]
[-l|--left] [-r|--right] [-n|--normalizeweight]
[--no-normalize] [--removeloops] [--normform]
[(-p|--prune) <prune>] [(-d|--determinize) <determ>]
[(-t|--train) <train>] [(-x|--xform) <xform>]
[--training-deriv-location <trainderivloc>]
[--conditional] [--no-deriv] [--randomize]
[--timedebug <time>] [-y|--print-yields]
[(-k|--kbest) <kbest>] [(-g|--generate) <krandom>]
[-c|--check] [(-o|--outputfile) <outfile>] infiles1
infiles2 ... infilesN
```

If you see something totally different, consult the README file that came with Tiburon (it should be in the same directory as Tiburon) for help before proceeding.

Now that Tiburon is installed properly, let's try using it. At its most basic, you can use Tiburon to make sure your RTG syntax is correct, because Tiburon usually expects an RTG as its input. Create the text file `rtg1`, exactly as above. Make sure to put a carriage return after the last line! Now type the following at the command prompt (all commands assume the files you write are in the same directory as your unpacked Tiburon files. If this is not true, add appropriate paths when executing commands):

```
% tiburon rtg1
```

You should see something like this:

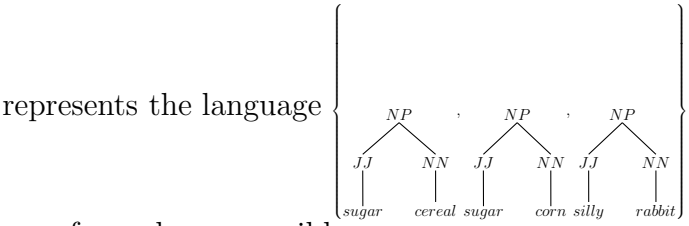
```
This is Tiburon, version 0.5.0
q4
q4 -> NP(q1) # 1.0000
q4 -> NP(q2 q1) # 1.0000
q1 -> NN(q0) # 1.0000
q0 -> boy # 1.0000
q0 -> girl # 1.0000
q3 -> the # 1.0000
q2 -> DT(q3) # 1.0000
```

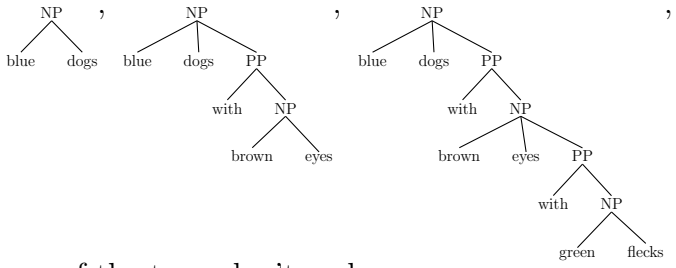
If you see an error message (and you have verified that Tiburon is installed properly, as above), your file format is probably incorrect. Make sure your file is typed correctly. If you still have problems, email `jonmay@isi.edu` with

a description of your problem including your file and the error message you received and we'll try and help you out.

Assuming everything worked correctly, take a look at the output. It looks sort of like the file you typed, but there are some differences. For one, the state names have changed – Tiburon automatically normalizes its state names, but semantically there is no difference. The rules might be in a different order, which also has no semantic difference from the input file. And there are "#" symbols and numbers. We'll explain those later on. Finally, the first line is missing. Any line or part of a line starting with "%" is regarded as a comment and ignored by Tiburon.

Exercises

- Write down 5 trees each in the languages represented by `rtg3` and `rtg4`
- Write an RTG that represents the language 

but *nothing else*. Use as few rules as possible.
- Write an RTG that represents a language of infinite trees. For example, you could represent 

etc. Don't worry if some of the trees don't make sense.
- Verify the RTGs you wrote are sensible by sending them through Tiburon, as described above.
- Write a CFG that represents the language $a^n b^n$, that is, a sequence like "a a a" that is some n letters long, followed by a sequence like "b b b" that is the same n letters long. You will need to make use of a special right-hand-side, `*e*`, which designates the *empty string*. For example,

the following simple CFG, `epscfg`, recognizes the language {`ab, b`} by making use of the `*e*` right hand side:

```
S
S -> A B
A -> a
A -> *e*
B -> b
```

It's often useful to intersect two RTGs to acquire the language common to both of them. For example, `rtg5`, below, could represent some possible translations of a foreign sentence:

```
%% filename: rtg5 %%
q
q -> S(w w w)
w -> likes
w -> John
w -> candy
```

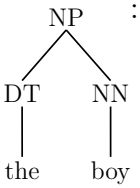
Some of the yields of the trees produced by `rtg5` are not good English sentences. `rtg6`, below, represents trees of several different English sentences:

```
%% filename: rtg6 %%
q
q -> S(subj verb obj)
subj -> John
subj -> Mary
subj -> Bill
verb -> likes
verb -> loathes
verb -> buys
obj -> trucks
obj -> candy
obj -> life
```

If you pass a sequence of several RTGs to Tiburon. Tiburon will form the intersection RTG of these input RTGs and return it to you. The intersection RTG is the RTG that produces exactly those trees produced by *all* of the

input RTGs. If there are no trees in common, the intersection RTG will just be empty (Tiburon's way of representing an empty RTG is a start state with no rules). In the case of the intersection of `rtg5` and `rtg6`, however, the intersection language consists of the single tree that represents a sentence that is both a grammatical sentence (i.e. it is in `rtg6`) and a possible translation (i.e. it is in `rtg5`):

```
% tiburon rtg5 rtg6
This is Tiburon, version 0.5.0
q0
q0 -> S(q2 q3 q1) # 1.0000
q1 -> candy # 1.0000
q3 -> likes # 1.0000
q2 -> John # 1.0000
```

It is easy to write a "degenerate" RTG that accepts only a single tree. Since any tree can be on the right side of a production, the following grammar suffices to represent the tree 

:

```
%% Filename: rtg7 %%
q
q -> NP(DT(the) NN(boy))
```

If you used this trick to do the exercises above, go back and do them again the "right" way! If you're having trouble try passing the tree into tiburon like so:

```
% echo ‘‘NP(DT(the) NN(boy))’’ | tiburon -
```

We may want to test if a certain tree is accepted by a certain RTG. We can use intersection for this as well:

```
% tiburon rtg1 rtg7
```

You should see a result that looks something like this:

```
This is Tiburon, 0.5.0
q0
q0 -> NP(q4 q3) # 1.0000
q1 -> the # 1.0000
q4 -> DT(q1) # 1.0000
q3 -> NN(q2) # 1.0000
q2 -> boy # 1.0000
```

If tree is not accepted by the RTG, the intersection will not contain any rules. For example, the derivative RTG `rtg7a` has no common trees with `rtg1`:

```
%% Filename: rtg7a %%
q
q -> NP(DT(the) NN(toy))
```

Try testing it!

Although there is no such thing as an "empty tree" corresponding to the "empty string" in string automata, there is use for an "empty transition" that serves to change state without consuming any symbols:

```
%% Filename: rtg8 %%
q
q -> S(subj was prednom)
subj -> John
subj -> Bill
prednom -> pp
prednom -> jj
pp -> PP(for the birds)
pp -> PP(with Mary)
jj -> happy
jj -> scared
```

Obviously, there are other ways to represent this language, but the empty transition can help reduce the size of a grammar and is handy for designers of RTGs.

Exercises

- Consider the following RTGs:

- `%% filename: rtg9 %%`

```
qe
qe -> A(qe qo)
qe -> A(qo qe)
qe -> B(qo)
qo -> A(qo qo)
qo -> A(qe qe)
qo -> B(qe)
qo -> C
```

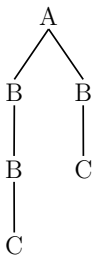
- `%% filename: rtg10 %%`

```
qa
qa -> A(qb qc)
qa -> A(qc qc)
qa -> A(qc qb)
qb -> B(qc)
qc -> C
```

  Write an RTG that represents the intersection of `rtg9` and `rtg10`. Then try using tiburon to calculate it. Is your answer the same? Are your answers correct?

- Determine which of the following trees are in the language represented by `rtg9`. Then write degenerate RTGs to check your answers.

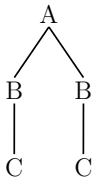
(a)



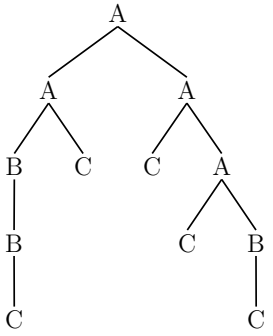
(b)

(c)

(d)



(e)



In the linguistic domain we want to represent preference of some trees over other trees. For example, perhaps "soft" is a more likely adjective than "quizzical" or a noun phrase with a determiner is more likely than one without. We can attach numerical weights to productions in an RTG to indicate this preference. For example, the above examples could be indicated in some RTG as:

```
...
jj -> soft # 0.8
jj -> quizzical # 0.2
np -> NP(dt jj nn) # 0.6
np -> NP(jj nn) # 0.4
...
```

A weighted RTG (WRTG) assigns a weight to each tree that it accepts. We normally think of these weights as probabilities, but it is not, strictly speaking, necessary for them to sum to 1. Trees not accepted by a language have a weight of 0 and those rules with a weight of 0 are typically omitted from an RTG representation. We associate a weight with each transition and multiply the weights of the transitions used to recognize a tree to obtain the weight of the tree. Here is `rtg1` as a WRTG:

```
%% Filename: wrtg1 %%
q
```

```
q -> NP(dt nn) # 0.6
q -> NP(nn) # 0.4
dt -> the # 1
nn -> boy # 0.8
nn -> girl # 0.2
```

The tree $\overset{\text{NP}}{|}_{\text{boy}}$ can be formed by using a rule with weight 0.4 and a rule with weight 0.8, thus it has a weight of 0.32.

Exercise: What are the weights of the other trees in this WRTG?

If a weight is not added to a rule, Tiburon assumes the weight is 1. This explains why Tiburon always adds `# 1.0000` to the RTGs it outputs. Tiburon treats all RTGs as WRTGs!

Tiburon can make the job of calculating the weight of trees recognized by a WRTG easier for you. Use the `-k` switch to list the k trees of highest weight, in order:

```
% tiburon -k 4 wrtg1
This is Tiburon, 0.5.0
NP(the boy): 0.4800
NP(boy): 0.3200
NP(the girl): 0.1200
NP(girl): 0.0800
```

Try it out with some of the examples we've already gone through. Since the examples we've presented so far are mostly unweighted, there won't be any particular order (i.e. all the weights will be 1) but as you can see this is a good way to ensure the RTG you've written has the trees you expect. Note that if you ask for more trees than are recognized by the RTG, Tiburon will print a warning. This shouldn't be a problem for some, like `rtg4`. In fact, you can determine just how many trees are in an RTG with the `-c` switch, like so:

```
tiburon -c wrtg1
This is Tiburon, 0.5.0
Check info:
        3 states
        5 rules
```

```
3 unique terminal symbols
4 derivations
```

Another fun way to see the trees in an RTG is to use the `-g` switch to list k random trees. This is a good way to see some less likely trees and possibly discover some errors in your RTG creation:

```
% tiburon -g 3 rtg4
This is Tiburon, 0.5.0
S(NP(the boy) VP(ran home)): 1.0000
S(NP(the clown PP(on NP(the clown PP(with NP(the monkey PP(with NP(the monkey PP(with NP(the monkey))))))))) VP(ran home)): 1.0000
S(NP(the clown PP(with NP(the boy PP(with NP(the clown))))) VP(ran home)): 1.0000
```

Sometimes it's useful to print the yield only. It tends to be easy on the eyes when the trees are large. For that we can use the `-y` option:

```
% tiburon -yg 3 rtg4
This is Tiburon, 0.5.0
the boy ran home: 1.0000
the monkey on the monkey with the clown ran home: 1.0000
the clown with the monkey on the clown ran home: 1.0000
```

We've been cheating when we talk about the printing of the trees and their corresponding weights in a WRTG. We're actually printing the trees associated with the WRTG *derivations* and their weights.

```
%% Filename wrtg2 %%
q
q -> S(subj vb obj) # 0.8
q -> S(subj likes obj) # 0.2
subj -> John # 0.7
subj -> Stacy # 0.4
obj -> candy
vb -> likes # 0.4
vb -> hates # 0.6
```

Consider the WRTG above. If you use `-k 6` you will see that there are 6 derivations, but only 4 unique trees:

```
% tiburon -k 6 wrtg2
This is Tiburon, 0.5.0
S(John hates candy): 0.3360
S(John likes candy): 0.2240
S(Stacy hates candy): 0.1920
```

```
S(John likes candy): 0.1400
S(Stacy likes candy): 0.1280
S(Stacy likes candy): 0.0800
```

Look at the sentence with yield "John likes candy". One derivation of this sentence has a probability of 0.2240 and the other has a probability of 0.1400. Both of them are lower than the derivation of the sentence with yield "John hates candy", currently at the top of the list, but the weight of the *tree* should be the sum of both derivations. We can use the `-d` option to *determinize* the WRTG, such that there is exactly one derivation for each unique tree:

```
% tiburon -d 5 -k 6 wrtg2
This is Tiburon, 0.5.0
Warning: returning fewer trees than requested
S(John likes candy): 0.3640
S(John hates candy): 0.3360
S(Stacy likes candy): 0.2080
S(Stacy hates candy): 0.1920
```

Notice, now there are only 4 derivations – one for each unique tree. Furthermore, we can see that `S(John likes candy)` is the most likely *tree* in this WRTG. Note that with the `-d` option you must specify the maximum time, in minutes, that Tiburon should work. This is because determinization can take a long time for very large RTGs and this gives you a way to quit if you only want to work for a little while.

Exercises

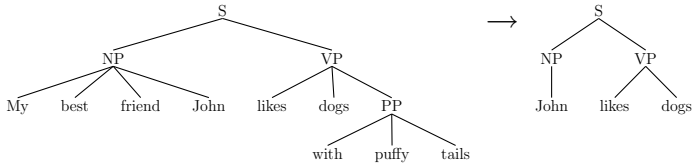
- Consider `wrtg3`, below:
  -
```
%% filename wrtg3 %%
q
q -> S(subj fought obj) # 0.8
q -> S(obj fought subj) # 0.2
subj -> Joe # 0.6
subj -> Steve # 0.4
obj -> NP(the volcano) # 0.7
obj -> subj # 0.3
```

Write down the 5 most likely trees in this WRTG. Use Tiburon to calculate the 10 most likely trees.

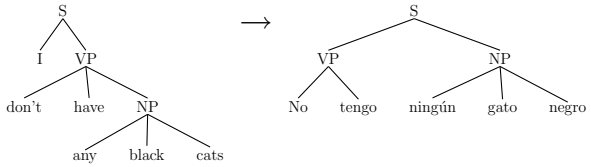
- Use tiburon to determinize `wrtg3`. Write down the determinized grammar and the top 10 trees in it. What changed?

4 Finite-State Tree Transducers

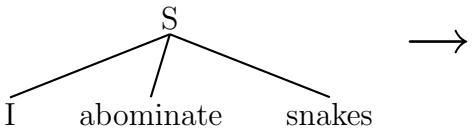
While an RTG lets you specify a set of trees, a transducer is a mechanism for changing trees into other things. Specifically, a transducer can change a tree into another tree, or it can change a tree into a string. There are lots of ways we might want to change trees. We might be interested in removing extra information from sentences:



We also might want to translate between languages:

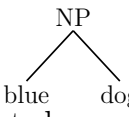


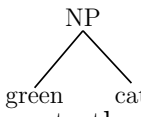
We may only have trees for one language. Hence, it becomes useful to change an English tree into, for example, a Japanese string:

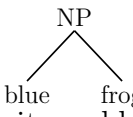


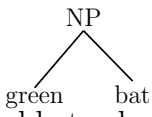
Finite tree-to-tree transducers (FTTT) look and work roughly the same way as RTGs. Consider the following very simple FTTT:

```
%% file: fttt1 %%
q
q.NP(blue dog) ->  NP(green cat)
q.NP(blue frog) -> NP(green bat)
```

Like for RTGs, a transducer has a start state and a set of rules. However, while an RTG had a state on the left and a tree on the right, an FTTT has a state *and* a tree on the left and a tree on the right. The left side specifies what the transducer should expect (i.e. which state it should be in and which tree it should see) and the right side specifies what to do. So the first rule could be interpreted "When in state `q`, reading the tree `NP(blue dog)`, write the tree `NP(green cat)`. Like we said, `fttt1` is pretty simple. All it can do is change 

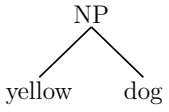
into 

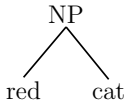
and tree 

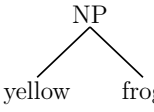
into 

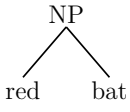
. If you presented any other tree to the transducer it wouldn't be able to change it. And if we wanted to change the transducer to be able to handle more trees we'd have to add a new rule for each input tree we wanted to handle. Fortunately, we can use *variables* to make the transducer more general. Let's rewrite the simple transducer as `fttt1a`, below.

```
%% file: fttt1a %%
q
q.NP(x0: x1:) -> NP(q.x0 q.x1)
q.blue -> green
q.dog -> cat
q.frog -> bat
```

The first rule might look a little weird. In particular, you may be wondering what `x0:` and `x1:` mean. They are *variables*. The left hand side of the rule, `q.NP(x0: x1:)`, means "When in state `q` and reading a tree headed with `NP`, with two children *about which I know nothing...*". We delay the transformation of these children for another rule, and in the right side of the rule specify in what position they should be placed, and how they should be handled. The right side of the rule means "...write an `NP`. For its left child transduce the subtree represented by variable `x0` in state `q`. For its right child transduce the subtree represented by variable `x1` in state `q`." So this transducer reads in tree headed with `NP` in state `q`, writes `NP`, and process each of its children in state `q`. It then allows other rules to transform blue to green, dog to cat, and frog to bat. As you can see, the same transductions are possible in `fttt1a` as in `fttt1` but now, if we want to also transform 

into 

and 

into 

all we have to do

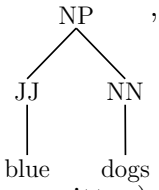
is add:

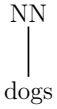
```
q.yellow -> red
```

and the transducer can handle two more input trees! The variables are indicated as variables by the colon (":") after them – by convention we use names like `x0` and `x1` but they can be any acceptable symbol name.

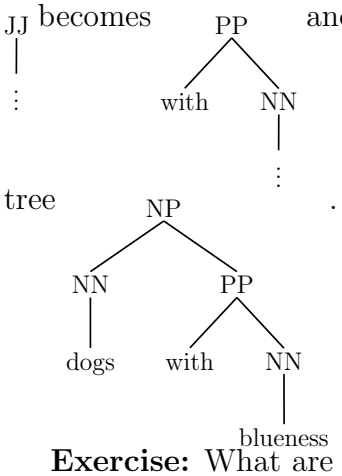
We can describe the variables a little more fully by putting a symbol on the other side of the colon. We can also use the transducer to make more dramatic changes to the input tree. The next example illustrates this:

```
%% file: fttt2 %%
q
q.NP(x1:JJ x2:) -> NP(q.x2 q.x1)
q.JJ(x1:) -> PP(with NN(adj.x1))
q.NN(x1:) -> NN(n.x1)
n.dogs -> dogs
n.cats -> cats
adj.blue -> blueness
adj.yellow -> yellowness
```

Let's look at the first rule in `fttt2`. At first glance it looks like some rules we've seen before, except for the `x1:JJ`. This simply indicates that the subtree represented by `x1` must have a root label of `JJ`. Also, notice that the variables on the left of the rule are in reverse order than they are on the right of the rule. This means the transformations we make to the left input subtree will appear on the right in the output tree, and vice versa. The first rule thus says, "when in state q, reading an NP with two children, the first of which is rooted with JJ, (and the second of which can be anything), write an NP, then for its leftmost child process the *second* child in state q, and for its rightmost child process the *first* child in state q." So if we apply this transducer to the tree 

, first the subtrees are reversed (because NP is read, but NP is also written). Then, in two transitions, 

is processed but unchanged.



and then blue becomes blueness, getting us the output .

Exercise: What are two other transductions that `fttt2` can perform?

Of course, rather than calculate this ourselves, Tiburon can do the job for us:

```
% echo "NP(JJ(blue) NN(dogs))" | tiburon - fttt2
```

The `-` is an argument that represents standard input. The tree is passed in through echo and pipe, and interpreted as the left most item in the transducer chain.. Tiburon should output an RTG. This is an example of the *application* operation. In general, a tree left-applied to a FTTT produces an RTG. Since the result is an RTG, we can do all the operations we did before, like `-k`, `-g`, `-d`, etc. in one step.

Exercises

- What would you expect the following command to produce? Try it!

  ```
  % echo "NP(JJ(yellow) NN(dogs))" | tiburon -k 1 - fttt2
  ```

- (More exercises...)

A finite tree-to-string transducer (FTST) looks just like a FTTT except the right side is a string, not a tree. This FTST is the tree-to-string version of `fttt2`:

```
%% file: ftst1 %%
q
```

```
q.NP(x1:JJ x2:) -> q.x2 q.x1
q.JJ(x1:) -> with adj.x1
q.NN(x1:) -> n.x1
n.dogs -> dogs
n.cats -> cats
adj.blue -> blueness
adj.yellow -> yellowness
```

Since the right side of FTST transitions is a string, we can make use of `*e*`, the special "empty string" symbol:

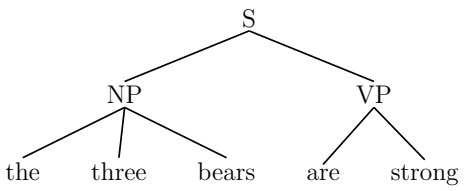
```
%% file: ftst2 %%
q
q.S(x1: x2:VP) -> t.x1 q.x2
t.We -> *e*
q.VP(x1:VP x2:) -> q.x1 t.x2
q.VP(x1: x2:) -> t.x1 t.x2
t.medicine -> medicina
t.sell -> vendemos
t.don't -> no
```

Just like for FTTT, you can apply trees onto the left side of FTST. However, where the application of a tree onto an FTTT produces an RTG, the application of a tree onto an FTST produces a CFG!

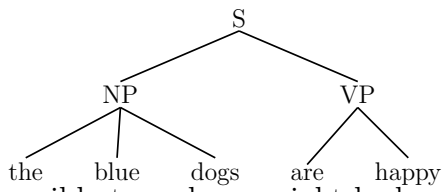
```
% echo "S(We VP(VP(don't sell) medicine))" | tiburon -k 1 - ftst2
```

Notice that the empty string token, `*e*`, is removed from the yield.

As you might expect, FTST and FTTT have weighted variants. The weights appear just as they do for WRTG. A single input tree may correspond to many possible output trees/strings. The multiplication of the weights of each rule used in performing a transduction gives the total weight of a transduction. Imagine we want to build a transducer that translates the following tree/string pairs:



→ los tres osos son fuertes



→ los perros azules están felices

A sensible transducer might look something like this:

```
%% file: wftst1 %%
q
q.S(x0:NP x1:VP) -> q.x0 q.x1  # 1
q.NP(x0: x1: x2:) -> t.x0 t.x1 t.x2 # 0.5
q.NP(x0: x1: x2:) -> t.x0 t.x2 t.x1 # 0.5
q.VP(x0: x1:) -> t.x0 t.x1 # 1
t.the -> los # 1
t.blue -> azules # 1
t.bears -> osos # 1
t.dogs -> perros # 1
t.are -> son # 0.5
t.are -> estn # 0.5
t.strong -> fuertes # 1
t.happy -> felices # 1
```

As you can see, there are multiple ways to translate "are" and multiple ways to reorder the NP. This is reflected in the weights of 0.5 for each possibility. Notice that many possible translations of the words and reorderings of the tree are not reflected in the transducer above. Unfortunately, in this case the incorrect translations have the same weight as the correct ones. In general there are many possible ways to transduce even simple sentences and a more useful translation transducer would have many more rules and be able to translate more sentences. Of course, as the complexity of the transducer grows it becomes harder for humans to set proper weights by hand. One of the more useful things Tiburon can do is automatically add weights to a set of unweighted rules to maximize the probability of a set of translations.

As an example, let's work with simple trees on the English side and construct a very basic, one-state transducer. The start state is thus entered into a new file:

```
%% file: wftst2 %%
q
```

In this example, we use trees that only have one internal symbol, X, and are strictly binary. A tree might be something like: `X(X(his associates) X(X(are not) strong))`. The translation sentence is "sus clientes están enfadados". Since the only nonterminal structure is binary, there are only two possible reorderings of the nonterminal structure:

```
q.X(x0: x1:) -> q.x0 q.x1
q.X(x0: x1:) -> q.x1 q.x0
```

Every English word in the sentence could translate to every Spanish word in the corresponding sentence, or could translate as the empty string:

```
q.his -> sus
q.his -> clientes
q.his -> estn
q.his -> enfadados
q.his -> *e*
```

Similar rules for all the other English words can be written. Using just one sentence pair is a bad idea, though, because there are lots of incorrect ways to translate this sentence pair correctly. We'll add the following sentence pairs:

```
X(Garcia X(and associates))
Garcia y asociados
X(X(Carlos Garcia) X(has X(three associates)))
Carlos Garcia tiene tres asociados
X(X(his associates) X(X(are not) strong))
sus asociados no son fuertes
X(Garcia X(X(has X(a company)) also))
Garcia tambin tiene una empresa
X(X(its clients) X(are angry))
sus clientes estn enfadados
X(X(the associates) X(X(are also) angry))
los asociados tambin estn enfadados
X(X(X(the clients) X(and X(the associates))) X(are enemies))
los clientes y los asociados son enemigos
X(X(the company) X(has X(three groups)))
la empresa tiene tres grupos
X(X(its groups) X(are X(in Europe)))
```

```
sus grupos estn en Europa
X(X(the X(modern groups)) X(sell X(strong pharmaceuticals)))
los grupos modernos venden medicinas fuertes
X(X(the groups) X(X(do not) X(sell zanzanine)))
los grupos no venden zanzanina
X(X(the X(small groups)) X(X(are not) modern))
los grupos pequeos no son modernos
```

Add all the possible word translation rules and we have a fully connected, but untrained, transducer. If the above sentence pairs are put in a file `train` then we use the `-t` flag to invoke Tiburon's training option:

```
% tiburon -t 20 -o wftst2.trained train wftst2
```

The `-t 20` option instructs Tiburon to run 20 iterations of the EM algorithm to set weights that maximize the probability of the sentence pairs in `train`. The `-o wftst2.trained` instructs Tiburon to write its output (in this case, the trained transducer) to the specified output file.

Training works on RTGs and CFGs as well as on FTTTs and FTSTs. Simply provide a file of trees or a file of strings instead of a file of tree-tree or tree-string pairs and the `-t` option will return an RTG or CFG weighted to maximize the probability of that corpus.

References

There is a wealth of material on tree automata, both highly theoretical and based in abstract algebra, and application-based. Below we list some resouces that have been useful.

The production of Tiburon is directly related to the research in tree automata being conducted at USC/ISI. To that end, the following papers may help to explain formally what is presented here informally:

> Jonathan May and Kevin Knight. "A better n-best list: practical determinization of weighted finite tree automata". In *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics (HLT-NAACL)*, New York, 2006.

Jonathan May and Kevin Knight. "Tiburon: a weighted tree automata toolkit". In *Proceedings, CIAA*, Taipei, 2006.

Jonathan Graehl and Kevin Knight. "Training tree transducers". In *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics (HLT-NAACL)*, Boston, 2004.

Kevin Knight and Jonathan Graehl. "An overview of probabilistic tree transducers for natural language processing". In *Proceedings of the Sixth International Conference on Intelligent Text Processing and Computational Linguistics (CICLing)*. Lecture Notes in Computer Science, copyright Springer Verlag, 2005.

Two sources cited quite often for comprehensive coverage of the theory behind classical tree automata are:

Ferenc Gécseg and Magnus Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest. 1984.

H. Comon and M. Dauchet and R. Gilleron and F. Jacquemard and D. Lugiez and S. Tison and M. Tommasi. *Tree Automata Techniques and Applications*. Available on `http://www.grappa.univ-lille3.fr/tata`. 1997.

Much of the literature regarding weighted tree transducers in the formal mathematics world focuses on *tree series transducers*, which are equivalent to weighted tree transducers but more convenient to use in a formal math setting (i.e. for proofs and closures). The following papers provide a good, if technical, foundation:

J. Engelfriet, Z. Fülöp, and H. Vogler. Bottom-up and top-down tree series transformations. *Journal of Automata, Languages and Combinatorics* 7. pages 11–70, 2002.

W. Kuich. Tree transducers and formal tree series. *Acta Cybernetica* 14. pages 135–149, 1999.