Types of Grammars

Kyle Johnson

Linguistics 401
26 January 2016

We left last time with two decisions. First, a component of our ability to recognize a grammatical sentence of English can be expressed solely in terms of the parts of speech that the words in that sentence belong to. We are able to express a certain kind of generalization in this way, then. We can express the fact that all of the sentences in (1), and many, many more, are grammatical with the statement that (2) is grammatical.

(1) a. The bad man smokes.
   b. The bad woman smolders.
   c. The big dog barks.
   d. A nasty elf dances
   e. Many horrible sentences confound.

(2) D(eterminer) A(djective) N(oun) V(erb)

And we decided that the syntax of English should be enabled to produce infinitely many, and infinitely long, sentences. So, our grammar must have a form that is capable of producing infinitely many, and infinitely long, strings of word types.

I introduced the idea that finite state automata could be the form that our grammars take. Finite state automata are virtual machines, or algorithms, that produce strings of symbols, and they are capable of producing an infinity of them. One way of describing a finite state automaton is as follows.

(3) a. There a finite number of states
   b. One state is designated as the initial state
   c. Some states are designated as final states
   d. There are transitions that connect any non-final state to any state.
   e. Each transition is associated with writing one symbol

As we saw, the automaton below produces the series in (4).

(4) a. The dog barked.
   b. The big dog barked.
   c. The big scary dog barked.
   ...

The states are numbered circles, and the transitions are represented by arrows connecting states. Each transition is associated with writing a symbol that represents one of our categories: “D” for determiner, “N” for noun, and so on. We could characterize in this way the series in (4) as the set of strings that are produced when this automaton goes through each of its states at least once. Each string you can imagine being paired with a particular “derivation.” To get “D N V,” you’d transit through the automaton passing through each state once. To get “D A N V,” you’d transit through state 2 twice.

Assignment one gave you some practice at using finite state automata to produce infinite series of strings of various sorts. If we were to use this formalism long enough, you’d begin to get a feel for how it expresses certain kinds of things. For instance, optionality can be expressed by allowing more than one transition from one state. And infinite additions of a string is expressed by a loop. Let’s just look at one more automaton to get our heads back in it.

What does this generate?

These automata have a set way of expressing relations. In fact, it is possible to talk about the “expressive power” of finite state automata: the kinds of patterns it can and cannot express. We might want to know if the expressive power of a finite state automaton matches that of human languages. That would be one way
of framing the question of whether the grammar we have come up with fits the learnability profile I described last week. Does the definition of a finite state automaton correctly limit the space of possible grammars to just those that children learn?

It's easy to see that finite state automata can describe grammars of a sort that no known human language is. For instance, the automaton in (5) produces strings of infinite length that always have an even number of "a"s (assuming that zero is even).

There aren't known syntactic processes that care about the even/odd distinction, so this isn't the kind of relation that human languages seem to exploit. If finite state automata are the right formalism for expressing syntax, then, we'll need to find additional constraints that explain why this particular kind of relation isn't found. We want to make sure, in other words, that the expressive power of our model of syntax fits the expressive power of syntax. One of our goals is to understand what a syntax can and cannot do, and we should make sure that our model confers this understanding.

There are also things that a finite state automaton cannot produce. One of these is an infinite string of symbols, $a$ and $b$, such that there are $n$ $a$s followed by $n$ $b$s. A finite state automaton cannot "remember" how many symbols it makes. In fact the "memory" of a finite state automaton is meager. The only kind of "memory" that can be encoded into any given state is what can be traversed by a finite series of states. So we should make sure that the resources that the syntaxes of language use match those of a finite state automaton.

With this brief sketch of the expressive power of a finite state automaton, let's continue our exploration of the syntax of English. We've seen that the series in (6) make up part of the strings of grammatical English sentences.

\[
\begin{align*}
\text{(6)} & \\
& a. \text{ D A}^* \text{ N V} \\
& b. \text{ D N (P D N)}^* \text{ V (P D N)}^* \\
\end{align*}
\]

Understand "X*" to mean any number of Xs, including none at all. There are very many more such strings to consider, of course. Note that in (6), there are two "PDN" strings that can cycle indefinitely. They are kinds of units, and they seem to be the same kind of unit. We will call these units: phrases. In the languages generated by finite state automata, phrases correspond to a sequence of states whose transitions produce the same sub-string.

There are also some that have properties that teach us something about the general form of the grammar we should be building. One of these illustrates a kind of action at a distance that we will need to find a way of expressing. This is the relationship between the word either and the word or that is illustrated by (7).

\[
\begin{align*}
\text{(7)} & \\
& a. \text{ She either ate or drank.} \\
& b. * \text{ She either ate and drank.} \\
& c. * \text{ She either ate.} \\
\end{align*}
\]

Let's see if we can write the rule that correctly pairs either with or. We'll start with a very simplified version of this problem. Let's consider only sentences that start with the noun she have as their second word either and then continue with a verb.

Let's begin by considering how to generate the series indicated in (8).

\[
\begin{align*}
\text{(8)} & \\
& \text{She either ate or drank in the bar.} \\
& \text{She either ate in the living room or drank.} \\
& \text{She either ate in the living room or drank in the bar.} \\
\end{align*}
\]

We can capture this with (9).
The P+D+N sequences in this series are called Prepositional Phrases. What this automaton does is generate a "either"+V sequence, then offers the possibility of producing any number of Prepositional Phrases before generating "or." After generating "or," it again offers the possibility of generating a V followed by one or more Prepositional phrases before stopping, or just one verb and then stopping.

Now, let's consider this series in (10).

What we see here is that in addition to the PPs that can be mixed into the strings that come between either and or, we can also get sequences of that followed by a N and a V. The class of words that that belongs to is called a "complementizer." So what we have here are C+N+V sequences that are being mixed in. These are called "Complementizer phrases." We can add them to the automaton we've made in (9) to give us (11).
Okay, we've gotten a start on working out how to characterize the strings that can come between *either* and *or*. Those strings can contain complementizer phrases, and that means they can contain the N+V sequence that starts the sentences that we are modeling. That N+V sequence can host a *either...or* string as well:

(11)  a. She either said that he either danced or cried or announced that he left.
     b. She either said that he either danced at the party or cried in the bathroom or announced that he left.
     c. She either said that he either danced at the party before the event or cried in the bathroom after the event or announced that he left.

And the two strings that are on either side of this first *or* can themselves contain another instance of *either...or*.

(12)  a. She either said that he either danced or cried or announced that he left.
     b. She either said that he either danced at the party or cried in the bathroom or announced that he left.
     c. She either said that he either danced at the party before the event or cried in the bathroom after the event or announced that he left.

Now these are not beautiful sentences. In fact, they are also extremely difficult to understand. But I think with some time and effort, they can be recognized as grammatical sentences. What makes them bad, I conjecture, is our memory and not our syntax.

How would we modify our automaton to produce them? We need a way of generating a string that allows an indefinite number of *either* followed by an indefinite number of *ors*. But more importantly, we must have the number of *ors* and *either*s be the same. We need to generate strings of this type:

(13)  a. She either said that he either thought that I either danced or whirled or knew that I hated it or announced that he had left.
     b. She either said that he either claimed that I both danced at the party or whirled in the bathroom or knew that I hated it or announced that he had left.

But this is something that a finite state automaton cannot do! It requires the same “memory” that we've seen *a^n b^n* would require, and this is beyond what a finite state automaton can do.
We need a type of grammar that is more powerful. One that can do recursion, so can manufacture an infinity of sentences, but also has the ability to "remember" things. Before moving on to a more powerful formalism for expressing our grammar, let me first introduce another way in which finite state automata can be defined:

(15) A finite state automaton is:
   a. A finite set of rules of the form $\alpha \rightarrow A \beta$ or $\alpha \rightarrow A$, where:
   b. "$\rightarrow$" means "can be replaced by," and
   c. $\alpha, \beta, \ldots$ are non-terminals (and correspond to "states"), and
   d. A, B, C, \ldots are terminals (and correspond to "symbols").

Using this formalism, a grammar that generates $D A^* N V$, could look like (16).

(16) a. $\alpha \rightarrow D \beta$
   b. $\alpha \rightarrow D \gamma$
   c. $\beta \rightarrow A \gamma$
   d. $\beta \rightarrow A \beta$
   e. $\gamma \rightarrow N \sigma$
   f. $\sigma \rightarrow V$

I introduce this alternative way of defining a finite state automaton because it makes it easier to see how we'll change this to enable us to capture the "either...or" business. The grammar we'll use is what's called a context-free grammar. A context-free grammar is one that can produce all the strings that can be produced by a finite state automaton, but also several other kinds of strings, including ones of the $a^n b^n$ type. To get a context-free grammar, all we do is change the restriction in the definition of finite state automata to allow the rules to have more than one non-terminal on the righthand side of the arrow. I'll do that with (17).

(17) A context-free grammar is one made up of a finite set of rules of the form, $\alpha \rightarrow ab$ or $\alpha \rightarrow a$, where:
   a. "$\rightarrow$" means "can be replaced by," and
   b. $\alpha, \beta, \ldots$ are non-terminals (and correspond to "states"), and
   c. $A, B, C, \ldots$ are terminals (and correspond to "symbols"), and
   d. and $a, b$ are either terminals or non-terminals

So, for instance, the rules in (18) make up a context free grammar.

(18) a. $\alpha \rightarrow \beta \gamma$
   b. $\beta \rightarrow D \delta$
   c. $\delta \rightarrow A \delta$
   d. $\delta \rightarrow N$
   e. $\gamma \rightarrow V$

What are the strings that these rules generate?
I’ll define a context free rule system now that builds in some detail that is specific to our needs. We shall slowly modify this system too, but it’ll last longer than did the finite state automaton model that we started with.

(19) Phrase Structure Grammar
   a. A phrase structure grammar consists of a finite set of rules, each of which have the following form:
      i. Each rule rewrites a symbol into a finite string of symbols. This is represented by "$\rightarrow$”
      ii. Each rule rewrites exactly one symbol.
      iii. The symbols are category labels.

This definition assumes that a category label is both a terminal symbol and a non-terminal symbol. I’ve slightly changed things by allowing the righthand side of the arrow to have more than two symbols, but this, as it turns out, still defines a context-free grammar.

(19) builds into the context free rule system our notion of "category." We can think of every word in a sentence as coming from a particular rule that rewrites a category symbol as the word belonging to that category we have chosen to use. So we could add to our phrase structure rules mappings like those in (20).

(20) a. $D \Rightarrow$ the
   b. $D \Rightarrow$ every
   c. $N \Rightarrow$ cat
   d. $N \Rightarrow$ hatred
   e. $V \Rightarrow$ amuse
   f. $P \Rightarrow$ on
   g. $P \Rightarrow$ because
   h. $A \Rightarrow$ happy
   i. $C \Rightarrow$ that
We should think of the rules in (20) as just expressing the morphological relationship between particular words and their categories. They’re not really part of the Phrase Structure rules that build our sentences (though we could rejigger things so that they would become part of those rules).

Let’s take a few steps towards recasting our finite state automata into phrase structure rules. We can characterize Prepositional Phrases with (21).

\[(21) \quad P \rightarrow P \ D \ N\]

And sentences this way:

\[(22) \quad S \rightarrow N \ V\]

The recursively available PPs can be produced with:

\[(23) \quad V \rightarrow V \ P\]

We also had C+N+V sequences to deal with in our either…or problem. These are called “Complementizer Phrases,” and they can be captured with (24).

\[(24) \quad C \rightarrow C \ S\]

These can be found inside Vs as well, so we might adopt this rule too:

\[(25) \quad V \rightarrow V \ C\]

This predicts that there can be an indefinite number of Cs in a sentence, just like Ps, and we haven’t seen evidence for this yet. But let’s adopt this view temporarily and revise it if evidence isn’t forthcoming. And now we’re finally ready to solve the either…or problem. The rule in (26) will do that.

\[(26) \quad V \rightarrow \text{DisjD} \ V \ \text{Disj} \ V\]

\[\text{a.} \quad \text{DisjD} \rightarrow \text{either}\]

\[\text{b.} \quad \text{Disj} \rightarrow \text{or}\]

These rules, when combined with the others that make sentences, will correctly pair either with a following or.

There’s a way of representing sentences that summarizes the rules that create them. These graphs are often called “phrase marker trees,” or just trees. Here’s an example.

\[(27)\]

```
S
  N   V
    she   V   P
      stood P   D   N
            on   the   table
```

The recursive nature of Vs is illustrated by:

\[(28)\]

```
S
  N   V
    she   V   P
      stood P   D   N
            after  the  dance
                on  the  table
```

And here’s how a either…or thing looks:

\[(29)\]

```
S
  N   V
    she   DisjD
        V   Disj
            V   P
                after  the
                  dance
```

The either part is optional, so we reflect this in the rule:

\[(30) \quad V \rightarrow (\text{DisjD}) \ V \ \text{Disj} \ V\]
We also want the rule for DisjP to be able to put two PPs together:

(31) She ran down the street or up the stairs.

It cannot mix a "V or P" however:

(32) * She ran or down the street.

So we should add:

(33) P → (DisjD) P Disj P

It is also possible to any two words of the same kind, but not any two words of different kinds.

(34) a. She ran up or down the street.
    b. She ran or skipped up the street.
    c. She ran up the stairs or drainpipe.
    d. She knows any or all things.

(35) a. * She ran down or the street.
    b. * She ran down the or street.
    c. * She ran or down the street.

So, perhaps we can generalize the rule to:

(36) α → (DisjD) α Disj α, where α can be anything.