Syntax, part 1

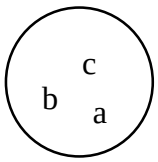
Michael Flynn
Carleton College

This section has two primary purposes. The first is to help readers feel comfortable with taking languages as purely *formal* systems, i.e. as sets of objects with no "meanings" at all. What's important about these languages is their *form*, not how they communicate anything, since, with an exception to be noted below, they aren't intended to communicate anything at all. We will see later that regarding natural languages like English as formal systems seems to lead to surprising and interesting conclusions about the nature of the human mind. The second goal is to introduce various techniques for handling formal systems like these, and a vocabulary with which to talk about their properties. Before too long we will use these tools to analyze human languages. In the meantime, I urge you to relax and enjoy the puzzles I have prepared for you.

1.1 Preliminaries

A *set* is an unordered collections of things.[1] Sets can be specified in many ways. For example, we might represent the set, which consists of the first three letters of the roman alphabet as in (1).

(1)



But this takes up a lot of space, and is expensive to print. A better way, for practical reasons, is to write down the set's *members* or *elements* within curly braces. This is

[1]We have the Russian mathematician Georg Cantor (1845-1918) to thank for the fascinating field called *set theory*. Among its wonders is that one can conclusively prove that, although there are exactly as many odd integers as there are integers, and just as many integers as there are fractions (i.e. rational numbers), there are fewer rationals than there are real numbers (i.e. numbers representable by perhaps nonterminating decimals), in reasonably straightforward senses of "as many ... as " and "fewer". In other words, there are (at least!) two "sizes" of infinity. For entertaining discussions of this point, among others, see Delong (1971) and Hofstadter (1979).

sometimes called the *list notation*. One way the set in (1) can be represented using this method is in (2).

(2) {a,b,c}

To say that a is an element of the set in (2) we write "a $\in \forall \in \forall$ {a,b,c}". If we were to give the set in (2) a name, say, A, we could write "a $\in$ A".

There are other ways the represent the membership of our set A, since the order of the presentation of the elements is irrelevant to the set's identity. So the set in (3) is just another way of writing the set in (2).

(3) {b,a,c}

We could also represent it as in (4), though this would be a bit perverse, since once we say that *a* is in the set, saying this over again doesn't change anything.

(4) {a,a,b,c}

(The set of colleges in Northfield, Minnesota consists of Carleton and St. Olaf, no matter how many times I reiterate that Carleton is in this set.) So here's two things to take to heart. First, the one and *only* thing relevant to the identity of a set is its membership.[2] Second, most, if not all, concrete representations of sets are misleading. Sets are too abstract to display with perfect fidelity.

Here's a bold move, laden with consequences we won't be able to explore here: We say that sets are things. One immediate consequence of this is that sets can be members of other sets. For example, the set {a, b, {a}} is a set that has three members: a, b, and the set that contains a. How many members does {{a,b,c}} have? (If you are not perfectly comfortable in answering "one", stop here and convince yourself.)

One way to specify sets that is less misleading is to use what is sometimes called the *predicate notation*. Here, one uses a predicate, for example, *be one of the first three letters of the roman alphabet*, to characterize the set's members. This is written as in (5).

(5) {x | x is one of the first three letters of the roman alphabet}

(5) is read "the set of all x such that x is one of the first three letters..." More generally, if P stands for any property, {x | x has P} will specify a set.[3] In a more technical notation,

---

[2]This is sometimes called the *Axiom of Extension*.

[3]This is sometimes called the *Axiom of Abstraction*. One may be jittery about this innocent sounding principle. For it's pretty easy to think of properties whose corresponding sets seem a bit dubious. Consider, for example, the set **R** = {x | x is not a member of itself}. Members of **R** include perfectly respectable sets such as the set of books in the Carleton College library (which is not itself a book, of course). The set of pizzas in Northfield (at 10:00 pm yesterday), the set of students enrolled in Linguistics 110 (now), and the set of authors of *The Sound Pattern of English* all enjoy membership in **R**. However, asking about and trying to determine whether or not **R** itself is a member of **R** leads to a bit of vertigo. Never fear. See any introduction to set theory for a way out of this pickle.

we might write equivalently {x | P(x)} to be read as "the set of all x such that the property P holds of x". Obviously, when sets get large, one will be inclined to make use of the predicate notation, and when one wants to specify an infinite set, this notation or something much like it will be mandatory, given the unfortunate finiteness of life.

What we might call the *alphabet* (in a kind of extended technical sense) for the writing system for the English language is a set of characters consisting of the letters a through z, the space, and a few punctuation marks.[4] A *string* over this (or any other) alphabet is an ordered sequence consisting of members of the alphabet. (6) contains six distinct strings over the alphabet described in this paragraph.[5]

(6) a. zzzzzzzz
b. zzzz
c. dog
d. god
e. m
f.
g. age often turns fire to placidity

For strings, though not for sets, order and repetitions count in determining the string's identity. So (6c) is distinct from (6d) (even though {d,o,g} and {g,o,d} are identical), and (6a) and (6b) are also distinct. (6e,f) raise a sometimes troublesome point that we may as well get used to now. As we said above, the space is a character in the alphabet we're currently considering. Even though it is invisible, it is *not* nothing. (The space, for example, has an ASCII code[6], but nothing, of course, doesn't.) Because of this, it's impossible to tell the exact identity of our strings in (6). To alleviate this problem, we might invent special symbols that serve to help us find the borders of our strings. For example, we might write (6e,f) as in (7a,b).

(7) a. # m#
b. # #

Notice that # is *not* in any of the strings we're trying to specify. It is a symbol that we use to help us identify strings when they are written down, nothing more. It's still a little hard to see what string we're trying to identify in (7b). To make such identifications easier, we might invent a symbol (*not* one from our current alphabet) and use it to mark a space. To give this maneuver the air of respectability it surely deserves, I'll choose the Greek letter sigma: σ. On this easier-to-see version, (6e,f) look like (8a,b).

(8) a. #σσm#

---

[4] I set aside the capital letters, and also the sensible observation that the space might itself be regarded as a punctuation mark, as might be things like paragraph breaks, and other conventions that indicate a writer's intent.

[5] (5g) is the first sentence in Stephen Jay Gould's essay "A Biological Homage to Mickey Mouse" reprinted in his (i.e. Gould's) collection *The Panda's Thumb* (W.W. Norton and Company, 1980).

[6] ASCII stands for *American Standard Code for Information Exchange*. It is a system, which assigns every letter, number, punctuation mark, and other symbols a special standard number, so that different computers and different programs can read each other's text. The ASCII code for the space is 32.

b. #σσσσσσσσσσσσσσσ#

This is sort of like staining a cell to reveal the mitochondria. It reveals, for example, that when I typed in the string (6f) I hit the spacebar fourteen times. We'll come back to this two-level structure for specifying languages a bit later. In the meantime, we'll assume that boundaries of all strings mentioned coincide with beginning and end of the visible portions of the string, and will therefore suppress the indication of string boundaries.

In a general, technical, and a bit misleading sense, we can think of a *language* as being a set of strings over some alphabet. (We make no restriction on the size of this set. Most of the time, it will be infinitely big.) Suppose we select the alphabet A (we might also call this the *vocabulary*) to be the set {a,b}.

A = {a,b}

Now consider the set of all strings over this vocabulary. (The set of all strings that can be formed by *concatenating* , i.e., stringing together, elements of some vocabulary V is sometimes called *the free monoid on V* (Wall (1972: 166).) It will consist of strings such as those in (9).

(9) a
b
aa
ab
ba
bb
aaa
aab

etc.

Obviously, the free monoid has infinitely many strings in it, since if I give you a string of length *n*, you can give me a string of length *n+1*. However, each string in this set is of finite length. Notice, then, that one can have languages of infinite size even though each string in the language is finite.

A *subset* B of a set A is a set such that every member of B is a member of A. (Is every set a subset of itself?) So, for example, if B = {a,b} and A = {a,b,c}, B is a subset of A, but A is not a subset of B. To express this in symbols, we write: A $\not\subset \forall\not\subset\forall$ B and B $\subseteq \forall\subseteq\forall$ A. A set B is a *proper subset* of a set A (written B $\subset \forall\subset\forall$ A) if B $\subseteq$ A and B$\neq$A.

The *empty* (or *null*) set, symbolized ø, is a set that contains nothing. (One could represent the null set like this: { }. But no one ever does this, except in contexts like the present one.) However, even though it is empty, it, like the space, is not itself nothing. The empty set is a perfectly legitimate object. For example ø $\neq$ {ø}, since the former contains nothing, but the latter contains something, namely the empty set. The empty set is a subset of every set. This is a bit counterintuitive, but one way to see it is to look at

from a negative point of view. When is a set A *not* a subset of a set B? Well, A is not a subset of B when A contains something that isn't in B. Since ø doesn't contain anything at all, it can't be the case that ø has something that a set B doesn't, no matter what set we take B to be.

Exercise 1

True or False?

a. $\{a\} = a$
b. $\{ø\} \subseteq \{\{ø\}, ø\}$
c. $\{a, b, \{c\}\} \in \{Ø, \{a, b, \{c\}\}\}$
d. $\{a,b\} \in \{a,b\}$
e. $\{a,b\} \subseteq \{a,b\}$
f. $ø \in ø$
g. $ø \subseteq ø$

The purpose of this section has been to introduce the very general notion of a language as a set of strings. At this point, you should understand this idea and also see how it might not be unreasonable to regard languages like English or Pashto[7] as sets of strings of words.

1.2 Generating Languages

Subsets of the free monoid mentioned in (9) will also be languages, since each of these subsets will also be a set of strings. We can pick out (or, to use a more technical term, *generate*) one or another of these subsets by giving a criterion by which to recognize a member of the subset that we're interested in. For example, consider the language $\mathbf{L_1}$, defined as in (10).

(10) $\mathbf{L_1}$: Vocabulary = {a,b}
Criterion: A string *s* is in $\mathbf{L_1}$ if and only if the first character of *s* is an *a*.

Imagine a device embodying these characteristics inspecting candidate strings and then reaching a decision, yes or no, to the question of whether or not the candidate is a member of $\mathbf{L_1}$. In this case, the first character must be an *a*, and each of the other characters must be either an *a* or *b*. If, for convenience, we restrict candidates to members of the free monoid described in (9), we get the following results.[8]

(11) a
*b
aa
ab
*ba

[7] Pashto is spoken in Pakistan and Afghanistan
[8] Of course, any candidate that isn't in the free monoid in (9) will be rejected.

*bb
aaa
aab

etc.

The prefixed asterisk means that the candidate is not in the target language.

Let's pause for a moment to ask a more "psychological" kind of question, by anthropomorphizing a bit. Imagine that we're introduced to a computer, which "speaks" a particular language, in other words, among other things, the computer will be able to tell, for any candidate string we propose, whether or not that string is in the machine's language. Perhaps some readers of this text have experienced this first hand. If you type an instruction to a computer, which it doesn't recognize, it will usually let you know in no uncertain terms, as in the following example I created many years ago on a computer which “spoke” Digital Command Language:

```
$ sek carls
%DCL-W-IVVERB, unrecognized command verb - check validity and spelling
 \SEK\
$
```

This example might seem quaint by today’s standards, since most people don’t interact with machines in this way anymore. But the point here is that the machine contains some information which specifies the words and syntax of a language it uses to interact with the outside world, and if you tried to interact with it in any other way, it would tell you something like ‘what you just said to me was jibberish, try again if you dare’.

If we were interested in the machine's linguistic capacities, we could adopt two closely related goals. The first is to discover what language the machine recognizes. To do this, we might propose various strings to the machine and see what happens, carefully noting which are accepted and which rejected, and then try to write a grammar of the language, which, to the extent that we get it right, should correspond in some fairly straightforward way to something that is actually in the machine.[9] Notice that it wouldn't be at all helpful at this juncture to simply go over to the CMC and start taking the machine apart, looking for the grammar of its language, since, in terms of all the little gizmos and whatnots inside the machine, we don't have the faintest idea what we're looking for. Likewise, if I'm interested in your linguistic capacities, it won't help very much to carefully remove the top of your skull and go rummaging around in the hills and valleys of your cerebral cortex. I need to know a lot more about what I'm looking for and how it's likely to be represented in there. In other words, our characterization of the language will be *abstract*, in the sense that we will focus on a disembodied system, a system we want to describe independently of the hardware it happens to be instantiated in. So one goal is to give an abstract characterization of the language "known" by the machine.

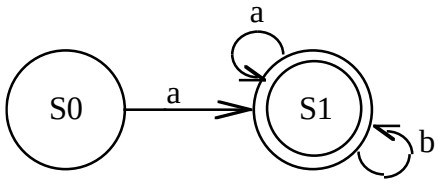
[9] Naturally, in the case of artificial machines and languages, there are people around whom we could simply ask, a route that, so far as I know, isn't available in the case of human beings and the languages they speak.

The second goal we may wish to adopt is to specify what a machine has to be like in order to handle the languages that it does. To adopt this stance is to shift the focus of the inquiry away from the languages for their own sake and toward the machines, which acquire and use them. Once we get a reasonable description of the language our machine speaks, we might ask what other languages it could have spoken had it only had the appropriate sorts of interactions with its environment. We might discover that languages fall into classes, and some machines handle some classes easily but that there are other classes that the machines don't recognize very easily or maybe can't recognize at all. To focus on this sort of thing would be to use the languages as window into the nature of the machines. We might say, this kind of machine recognizes such and such kind of languages very easily, but can't manage these other sorts of languages. Presumably this would be so because of how the machines are *designed*, that is to say the nature of their structure before we give them any language at all. This maybe won't be such a fascinating adventure in the case of our computers, but for humans and their languages, well, it's a different story, as we will see.

Returning to our example in (11) above, supposing that the computer's language is infinite, we will only be able to make a reasonable guess at the identity of the language, but that will be good enough for us. Also, since the computer itself is finite, the "program" that we propose for its "mental" computations will have to be finite, since, of course, it is instantiated in the finite computer. Suppose that we typed in the candidates in (11) and received the responses indicated there, i.e. the computer responds "yes" to *a*,"no" to *b*, "yes" to *ab*, etc. What we do now is write some kind of a flow chart that mimics the computer's responses. We'll then take what is sometimes called a "realist" stance (more on this later), and attribute to the machine the properties of our model.

Here's one way of representing a model that will do the job in this case. These systems are sometimes called *finite state automata*, a notion we will make more precise in a moment. Consider then an automaton that will generate the language $\mathbf{L_1}$ of (11).

(12)



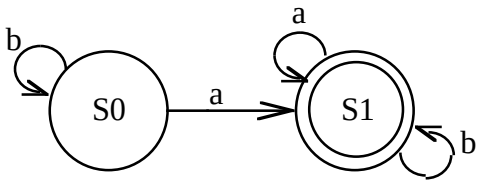
Here's how to interpret this diagram. The circles represent *states* of the machine. Concentric circles indicate special states, called *final states*. The lines and loops, called *arcs*, represent instructions on what the machine should do given a particular input. The machine always starts in an initial state, S_0, and contemplates the first (i.e. leftmost) character of the candidate string. Suppose our machine is looking at the string *aba*. Since the first character matches the label on the arc leading away from S_0, the machine "accepts" the *a*, switches into state S_1, and examines the next character in the string, which in our example is *b*. This matches the label on one of the loops leading away from S_1. Therefore, the machine follows the loop, accepting the *b*, returns to S_1, and examines

the next character in the string, an *a*. Now the machine follows the other loop, accepting the *a* and returning to S_1. At this point the machine runs out of candidate string. The rule in this case is that since the machine is in a final state, the candidate is accepted. The machine then flashes "yes" on the monitor, and waits for the next candidate.

Suppose now we type in *ba*. The machine starts out in S_0, and inspects the *b*. But there is no arc leading away from S_0 labeled with a *b*. So the machine cannot accept the *b*, and will then stop and print "no" on the monitor.

Let's consider a slightly different automaton, the one in (13).

(13)



What language will this automaton accept? First of all, it will obviously accept all the strings accepted by the automaton in (12), since (13) has all the paths that (12) has, and more besides. The new path is the loop labeled *b* on the state S_0. (13) will accept the string *ba*, since it can loop on the initial *b* back into S_0, switch to the final state S_1 on *a*, thus stopping in a final state. On the other hand, the machine will reject *bbb*, since though the machine will accept the entire string, it won't be in a final state at the end.

Let's describe finite automata (abbreviated *fa*) more generally. Fa's have a finite number of states, linked to each other by a finite number of labeled arcs. The arcs are instructions telling the machine what to do when it encounters a particular input , e.g. "accept it and move to state S_n". For the machine to recognize anything, at least one of the states must be a final state. There are no other restrictions. Fa's can have any (finite) number of initial and final states, and these can be connected by any (finite) number of arcs.

By specifying the fa that generates the language of the computer we were introduced to a few paragraphs back, we've made a proposal concerning the abstract characterization of the relevant portion of the machine's "mind".

We might digress here briefly to anticipate what relevance all of this will have to our primary objective, which is, as you'll recall, to describe human languages and the minds that "know" them. We can think of natural languages like English as (among other things) infinite sets of strings of words. For the moment, think of the words in English as atomic units. (We'll return later to investigate their internal structure.) So, *the, running, sleeps, baby, computer, have,* etc. are all items in the English vocabulary, analogous to the set {a,b} in our language $\mathbf{L_1}$ above. Suppose we took the set of words in any ordinary

dictionary, added in a few proper names, formed some strings over this set, and presented those strings to a person who knows English. We will likely get results like this[10]:

(14) a. babies sleep in cribs
b. *sleep babies cribs in
c. colorless green ideas sleep furiously
d. *furiously sleep ideas green colorless
e. do you often walk to school
f. *walk you often to school
g. the pen that I lost was expensive
h. I don't know where my pen is
i. *the pen that I don't know where is was expensive
j. how Ann Salisbury can claim that Pam Dawber's anger at not receiving her fair share of acclaim for *Mork and Mindy's*[11] success derives from a fragile ego escapes me.
k. *how Ann Salisbury can claim that Pam Dawber's anger at not receiving her fair share of acclaim for *Mork and Mindy's* success derives from a fragile ego escape me.

Putting aside for awhile the question of whether or not this point of view is particularly illuminating, we could regard our English speaker as being rather like the computer we considered a moment ago, in at least this respect: The person is a finite object, capable of deciding, for any given string over the English vocabulary, whether or not the candidate string is in English or not. We might then try to write a description of the mechanism the person possesses that accounts for this skill. Furthermore, we might find it convenient (or even necessary) to couch this description in abstract terms, that is, in terms that are independent of the person's "hardware" (e.g. neural organization and electro-chemical flows in the brain). In other words, we might end up with a description analogous in some ways to our theory about the internal organization of the computer. This is one of the central problems of linguistics theory, and we will have much to say about it later. For now, though, let's return to our investigation of finite automata.

(16) is a representation of an fa that will generate the language in (15).

(15) a
aba
ababa
abababa

etc.

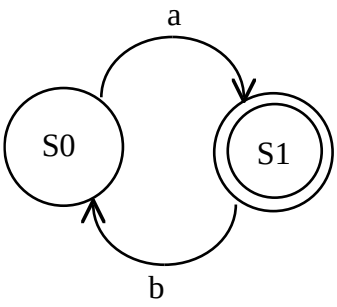
or more accurately and succinctly, $\mathbf{a(ba)^n}$, $n \geq 0$.[12]

---

[10] Examples (14c,d) are very famous examples from Noam Chomsky's first book, *Syntactic Structures* (1957). (14j,k) were cited by Lila Gleitman in her paper "Maturational Determinants of Language Growth" (1981). As Gleitman noted, (14j) originally appeared in a letter to *TV Guide*.

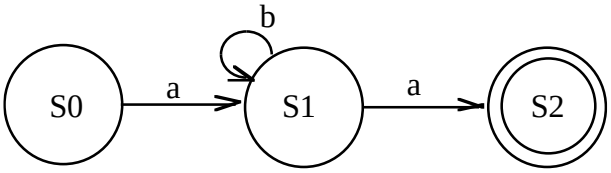
[11] *Mork and Mindy* was a wacky television sitcom from the late 1970's and early 1980's, starring Robin Williams (and of course Pam Dawber). If you don't believe me, go to http://www.sitcomsonline.com/morkandmindy.html

(16)



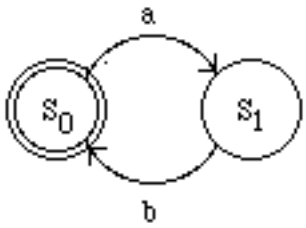
(17) is an fa that will generate the language $\mathbf{ab^n a}$, $n \geq 0$.

(17)



(18) generates $\mathbf{(ab)^n}$, $n \geq 0$.

(18)



Notice that (18) will also recognize the *empty string*.

Exercise 2

Every fa we've seen so far generates an infinite language. Of course, there are fa's that recognize finite languages. Give an example of one by modifying any of the fa's we've seen so far. Try to give a procedure to check, given any fa at all, whether or not that fa recognizes an infinite language.

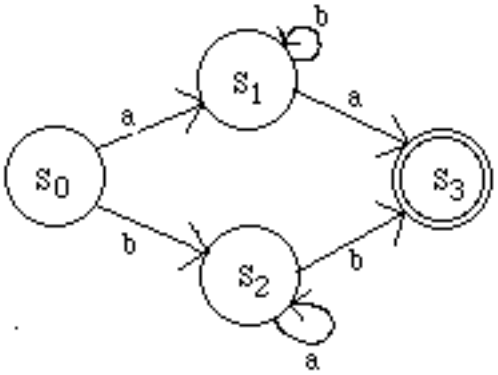
Exercise 3

Describe (in English) the language generated by each of the following fa's.

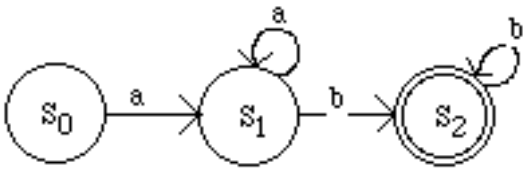
---

[12] The superscript *n* indicates that the sequence in parentheses, in this case, **ba**, can be repeated *n* times.

a.



b.



Exercise 4

Draw fa's for each of the following languages:

a. $\mathbf{aba^n c}$ $n \geq 0$.

b. $\mathbf{ac^m (ba)^n}$, $n,m \geq 1$

Exercise 5

Take words in English to be atomic units, analogous to the letters **a** and **b** in the examples above. (That is to say, suppose words have no internal parts.) Draw a fa that generates *exactly* [13]the language that consists of the following strings:

books have pages
some books have pages
books have many pages
this book has pages
this book has many pages

Exercise 6

Consider a language like the one in Exercise 5, except that sentences such as the following are in it as well:

this book has many many pages
this book has many many many pages
this book has many many many many pages

Suppose that arcs are costly, say, in terms of memory space in a computer. (Alternatively, you can imagine that I charge you ten cents for each arc in the fa you draw.) I give you a choice: You can either draw an fa for

a. the language in which *many* can be repeated as many as thirty eight times (this language will have finitely many sentences in it)

or

b. the language in which *many* can be repeated unboundedly many times (this language will have infinitely many sentences in it).

Say which language would you choose to draw an fa for, and why.

Exercise 7

[13]*Exactly* here means that the language generated consists of all the sentences given, and that the language contains no other sentences. We say that the fa you draw generates all and only the sentences given.

It's easy to draw an fa for the language $\mathbf{a^n b^m, n,m \geq 1}$. Do it. However, not only is it difficult, it's flat out *impossible* to draw an fa for $\mathbf{a^n b^n, n \geq 1}$ (i.e., where there must be exactly the same number of a's and b's in each string). Sketch out how you would draw an fa for $\mathbf{a^n b^n}$ for any finite **n**, and indicate what the problem is when **n** can get indefinitely large.

In this section we've seen how one can specify (or *recognize* or *generate*) an infinite language using a finite mechanism. We've also had a glimpse of how one might use the linguistic capacities of a thing (machine or organism) to approach the question of what sort of mechanisms has to be inside the thing. A finite state automaton is a particularly simple representation of a language recognizing capacity, so simple in fact that there are well defined languages which no fa, no matter how large, can recognize. (Thus a thing that recognizes one of these languages can't have (merely) an fa inside.)

1.3 The MIU system[14]

The MIU system generates a language, in the sense of "language" we have been using so far. We examine it solely for the purpose of extending our technical apparatus and conceptual framework. The vocabulary for the MIU system is {**M**,**I**,**U**}. The system has four rules.

Rule 1: If you have a string whose last letter is **I**, you can add a **U** at the end.

In more abbreviated form: **xI => xIU**

The **x** in this abbreviation is a *variable*. Of course, no string in the MIU system looks like "xI" since "x" is not even in the vocabulary. The "x" here is to be regarded as a variable over strings of symbols that are in the vocabulary.

This is perhaps a good time to introduce the important distinction between an *object language* and a *metalanguage*. If I talk about the mighty Carleton Knights, there's no danger of confusing the Knights with my talk about them. But if I'm talking about a language, I should be careful to distinguish the language I'm talking *about* (the object language) from the language I'm talking *in* (the metalanguage, i.e. the language used to talk about the object language). In this case, the object language is the one generated by the MIU system. The metalanguage, in the first version of the rule, is English. In the rule's abbreviation, the metalanguage is a special one that I use to shorten, and thus make more readable, the English version of the rule. Recall (6e,f) and their representations (8a,b):

(6) e. m
f.

(8) a. #σσm#

[14]This system was invented by Douglas Hofstadter and you can read about it and very many other interesting things in his book *Gödel, Escher, Bach: An Eternal Golden Braid* (Hofstadter 1980).

b. #σσσσσσσσσσσσσσσ#

The items "m" and " " (i.e. the space) are in the object language. But "#" and "σ" are not in the object language. They are in the metalanguage.

We will have much to say about metalanguages later on. For now, though, it's worth observing that it is easy to change the metalanguage dramatically while leaving the object language alone, just as snow falls in Northfield no matter how we choose to talk about it. For example, Rule 1 says that if I have a string such as **IUMMI**, I can form a new string in the language by adding a **U** on the end, i.e. **IUMMIU**. However, if I were writing for a Dutch speaking audience, most likely I would have chosen a different metalanguage, namely Dutch. Here's Rule 1 in that language:

Regel 1: Als je een rij hebt met als laatste letter een **I**, dan kun je een **U** aan het eind toevoegen.

Afgekort: **xI => xIU**

This change might seem quite dramatic, but the important point to note is that the object language doesn't change at all. This rule still adds a **U** to any string that ends in an **I**. For now, we will regard the choice of metalanguage as a matter of convenience, and we will feel free to modify metalanguages at will, changing them to suit our purposes or whenever the spirit moves us. Later on, we will see that the choice of a metalanguage for describing natural languages such as English, Tamil, and Hausa in linguistics is crucial, and can in fact be thought of one of the central problems for the discipline.

Let's now return to the MIU system.

Rule 2: Suppose you have **Mx**. Then you may also form **Mxx**.

Abbreviation: **Mx => Mxx**

This rule will take **MIUU** into **MIUUIUU**. It takes everything to the right of the initial **M** and adds a copy of that to the right of the original string. One thing to notice here is that since no variable appears to the left of the **M**, the **M** must be the *first* symbol in the input string. So this rule will not apply to a string like **IMU**.

Rule 3: If **III** occurs anywhere in a string, this *substring* (i.e. part of a string) may be replaced with a **U**.

Abbreviation: **xIIIy => xUy**

Rule 3 will change, for example, **MIIIUI** into **MUUI**. Notice here that I have chosen two distinct variables, x and y, to indicate that there may be substrings on either side of the three adjacent **I**s. I don't want to require that these substrings be identical, which I would imply if I had written **xIIIx => xUx**. The intention here is that the substring flanking the target (i.e. the material to be affected by the rule, in this case, **III**) may be identical, but they needn't be. So this rule could apply to **MUIIIMU**, changing it to

MUUMU. If I had written the rule with two **x**'s flanking the target, then this is the only kind of input the rule could apply to. Also it is to be understood that either of the variables (or both) could be *null*, that is, there needn't be a substring that the variable is standing for. For example, Rule 3 will apply to **IIIUM**, changing it to **UUM**.

The last rule is

Rule 4: If two adjacent **U**'s appear in a string, they can both be deleted.

Abbreviation: **xUUy => xy**

This rule would take, for example, **MUUI** into **MI**.

Notice that, given a string, there may be more that one way for a rule to apply to it. To describe this, let's introduce a bit more terminology. Let's call the specification of possible inputs to a rule, the left side of the arrow in the abbreviations, a *structural description*. What a structural description does is pick out a class of objects to which the rule can apply. This is actually a commonplace concept. For example, some businesses offer discounts to people who are 65 years of age or older. In our terminology, "being at least 65 years old" would be the structural description of the rule which results in a discount, and we can speak of people meeting or failing to meet that description. Likewise, we can say that **MUUI** meets the structural description of Rule 4, but **MUIU** does not. The right side of the double arrow, the instruction that specifies what to do if the structural description is met, we can call the *structural change*.

In order to determine whether or not a string meets a structural description, we *factor* it. This simply means to divide up into its parts, but as in factoring numbers in arithmetic, there are usually many different ways to do this. Some factorizations may satisfy a given structural description while others may not. For example, consider the string **MIUUUI**. There are many factorizations of this string, some of which are given in (19) (I put a | between factors):

(19) a. **M | IU | U | UI**
b. **MI | UU | UI**
c. **MIU | UUI**
d. **M | IU | UU | I**
e. **MIU | UU | I**

We say, for example, that analysis (19a) yields four factors **M**, **IU**, **U** and **UI**. Analysis (19b) yields three factors, (19c) gives two factors, etc.

Suppose we read Rule 4 as specifying that in order for the rule to apply, strings must be factorable into three substrings, the first and last of which can be any string at all (including the null string) while the middle substring must consist exactly of two **U**s. On this reading, analyses (19b and e) will both satisfy the structural description of the rule, but the others will not. So, technically speaking, our rules apply to *strings under an analysis*, not just strings.

Exercise 8

Consider the string **MUUIUUI**. Give all the factorizations that meet the structural description of Rule 4, and in each case give the result of applying the structural change. Do the same for the string **MUUUI.**

I now repeat the rules of the MIU system here in their abbreviated form:

Rule 1: **xI => xIU**

Rule 2: **Mx => Mxx**

Rule 3: **xIIIy => xUy**

Rule 4: **xUUy => xy**

Of course, so far this system doesn't generate anything at all, since all of these rules have the form of *if - then* statements. You can't apply any of these rules until you have a string to apply them to. We need a starting point, which we will call the initial string. Here it is: **MI**.

We can think of this system as licensing *derivations*. A derivation is a demonstration that a particular string is in fact in the language generated by the system. The form of a derivation is a sequence of lines, each of which follows from the one above it by one of the rules, except for the first line, which is always the initial string. We say a given derivation is a *derivation of* the last line in the sequence.[15] So in the MIU system, every derivation begins with **MI**. Then we apply one or another of the rules in order to produce a new string, to which we can again apply one or another of the rules to produce another new string, etc. Each string so produced is shown to be generated by the MIU system.

Here's some derivations in the MIU system. (To make derivations easy to check, we'll adopt the convention that we indicate next to a line the rule, which was used to derive it from the line above.)

(20) a.

1. **MI**	initial string
2. **MIU**	(1)
3. **MIUIU**	(2)

b.

1. **MI**	initial string
2. **MII**	(2)
3. **MIIII**	(2)
4. **MIIIIU**	(1)
5. **MIUU**	(3)

[15] Actually, derivations have different forms in different systems We will examine some variations on this theme shortly.

c.
1. **MI** initial string
2. **MII** (2)
3. **MIIII** (2)
4. **MIIIIIIII** (2)
5. **MUIIIII** (3)
6. **MUUII** (3)
7. **MII** (4)

There are several things to notice. The rules may apply in any order (so long as their structural descriptions are met, of course). If two or more rules are applicable to a given string, any of the rules may apply, but they must apply one at a time. It sometimes may happen that the application of one of the rules may destroy the environment for the other to apply. This often happens in the description of natural languages, and linguists call such an order of rules a *bleeding order*. If a rule applies so as to make it possible for another rule to apply, we call it, you guessed it, a *feeding order*. For example, both rules 1 and 3 will apply to **MIII**, but application of rule 3 will destroy the chance for rule 1 to apply i.e. rule 3 will *bleed* rule 1 in this case. Derivation (20c) is a bit perverse, as it derives **MII** in seven steps, when it could have been derived in two, as the derivation itself shows. Perverse or not, however, it is a perfectly fine derivation. It may be stylistically inept, but it is nevertheless a legitimate demonstration that **MII** is generated by the system. This example also shows that there in general is more than one derivation for each string, so there is no such thing as *the* correct derivation of a string.

As one gets good at doing derivations, it is tempting to collapse steps. For example, for (20b) an experienced MIUer might be inclined to write:

1. **MI** initial string
2. **MIIII** (2,2) (for two applications of Rule 2)
3. **MIUU** (1,3)

There's nothing really wrong with this short hand, except that such derivations can be difficult for less experienced players to read. So, in the interest of politeness, we hereby make such abbreviated derivations illegal. When you are asked to give a derivation, as in exercise 8, write out every step.

When doing derivations, it is sometimes helpful to work from both ends. For example, suppose I am asked to derive **MIUIUIUIU**. I might not see how to get this right off, so I might reason as follows. I could derive this string by rule (2) if only I could derive **MIUIU**, so my problem now reduces to deriving this string. Thus I have so far:

MI

MIUIU
MIUIUIUIU (2)

I see that **U**'s are introduced by rule 3. Thus, I apply rule 3 "backwards" to the current line twice:

MI

MIIIIIII
MIUIII (3)
MIUIU (3)
MIUIUIUIU (2)

Now it's clear that I can get to **MIIIIIIII** by repeated applications of rule 2:

MI	initial string
MII	(2)
MIIII	(2)
MIIIIIII	(2)
MIUIII	(3)
MIUIU	(3)
MIUIUIUIU	(2)

Exercise 9

Show that the following strings are generated by the MIU system, by displaying derivations for them.

a. **MUI**
b. **MIIUU**
c. **MUUUI**

Suppose that I gave you the string **UIM** and asked you to decide whether or not this string is generated by the MIU system. One thing you could do is sit down with a pad of paper (better make it a big one!) and start doing derivations, hoping that sooner or later (hopefully sooner) the string will show up and you can triumphantly report "yes". Nevertheless, I venture to think that no one, at least no one who has done exercise 9

would dream of resorting to this tactic, since it's plain that this string will *never*, *ever* show up, no matter how long you sat doing derivation after derivation.

It's useful to formulate explicitly how we know that **UIM** will never appear as a line in a legal derivation. The trick, of course, is to stop working *within* the system and instead look *at* it. In this case, we consider rule 1, and observe that if the input to this rule has an initial **M**, the output will as well. Rule 1 is "initial **M** preserving". Rule 2 requires an initial **M** as a part of its structural description, and returns a string that maintains it. So it is "initial M preserving" too. It's easy to see that rules 3 and 4 also have this property. Since the one and only initial string has an initial **M**, and all of the rules are "initial **M** preserving", it follows that *every* string generated by the MIU system will have an initial **M**. From this general theorem, the fact that **UIM** is not among the strings generated by the system follows as a trivial corollary.

The preceding paragraph must seem like an exposition of the obvious, but in other cases it is not always so clear whether or not a given string is generable by a given system. When faced with such a problem, it's often a good strategy to first work within the system for awhile hoping that you suddenly see how to generate the string. If this doesn't work, the next thing to try might be to think of a property the string has that you can show is possessed by no string generable by the system. That would tell you that the string is not generable. The more complicated systems get, however, the harder it is to tell whether or not some strings are generable by those systems. In some cases, in fact, one can show that there is no procedure that will work every time. Further discussion of this would take us too far afield, but those who are intrigued might begin to investigate this with one of the math or logic books mentioned in the bibliography.

Exercise 10

Say why **MUIM** is not generable by the MIU system. (As mentioned above, one way to do this would be to demonstrate something stronger than you need, and then observe that the result you want trivially follows from this. For example, one might try to show something about the number of **M**s that can appear in a string.)

Exercise 11

Consider the string **MU**. If it is generable by the system, give a derivation. If not, demonstrate this. (This problem may be a bit challenging. It is a rewarding exercise not to give up on it too easily. Give yourself a good chance to solve the problem. For discussion of it, see *Gödel, Escher, Bach*.)

Before going on to more interesting languages and techniques for describing them, let's pause to summarize some of the points mentioned in this section. We saw how languages could be thought of as sets of strings, and how we can precisely specify the membership of infinite languages in a finite way. This is useful, since natural languages like Hebrew can be thought of as infinite sets of strings of words, while the organisms that "know" these languages are finite. Finite State Automata are very simple systems for generating languages. We've also seen how one can generate infinite languages like the MIU system

by writing finite sets of rules. We distinguish between the language generated (the object language) and the language in which the rules are written (the metalanguage). Rules of the sort we examined have two parts: structural descriptions, which specify the class of objects to which the rules can apply, and structural changes, which specify what the rule does. Membership in a language can be demonstrated by a derivation, while nonmembership is most often demonstrated by constructing an argument from outside of the system, which is based on the nature of the rules.

Here's a list of some terminology that may have been unfamiliar to you before reading this chapter. Be sure you have a pretty clear idea of what each term means before you proceed.

set	*member*	*alphabet* (or *vocabulary*)
string	*language*	(*proper*) *subset*
free monoid	*generate*	*finite state automata*
variable	*object language*	*metalanguage*
substring	*structural description*	*structural change*
strings under an analysis		*derivation*

1.4 The Propositional Calculus

The languages we have considered up to now have been purely pedagogical devices. We have examined them for no other reason than to illustrate certain basic skills in manipulating systems that specify languages. We'll turn now to a language (actually, a class of languages) that has a more distinguished pedigree, even though our main interest is still in acquiring skills for building systems for describing the syntax of languages.

The Propositional Calculus (sometimes also called the Propositional Logic, the Sentential Calculus (or Logic), or the Theory of Truth Functions) was first discussed at some length by the Stoic philosophers, who flourished in Greece and nearby regions around the time of Aristotle (3rd and 2nd century B.C.). Much of this work was lost or ignored in Europe in the Middle Ages and through the Renaissance, and the system was reinvented by the German philosopher and mathematician Gottlob Frege in the latter half of the 19th century. Frege's presentation of the system in his *Begriffsschrift* (Frege 1879) is commonly regarded as the beginning of the modern era of the study of logic. (For very interesting accounts of these events, see Delong 1971 or Prior 1962.)

We will postpone for now consideration of why this system was regarded as so interesting, and concentrate instead on manipulating the syntax of it so as to acquire more tools for the analysis of natural languages. In fact, we will not consider Frege's syntax for the system at all, as it is rather awkward and doesn't play any role in current day linguistic analyses. We'll instead explore various deployments which are closely related to (and sometimes identical with) treatments that appear in many present-day logic textbooks, most of which descend from the systems as they were presented in *Principia Mathematica*, the great three volume work on logic and mathematics written by Alfred North Whitehead and Bertrand Russell and published between 1910 and 1913.

I mentioned a few pages back that linguists are keenly interested in the metalanguages that are used to describe object languages (object languages such as English, Chinese, Hungarian, etc.). Our goal here is to take the comparatively simple language of the propositional calculus and introduce a number of devices and techniques that may be helpful in explicating the structure of a wide variety of languages. Later on, we'll see how some of these devices and techniques can be applied to natural languages.

Another way in which the propositional calculus differs from the languages we've considered so far is that it has an *intended interpretation*. By that I mean that most of the symbols in the system are really symbols in the sense that they are meant to stand for something outside the language itself. We will informally introduce aspects of this interpretation as we go along, primarily because this will suggest why some of the symbols are named the way they are.

1.4.1 The Syntax of the Propositional Calculus by means of a Recursive Definition

The vocabulary of the propositional calculus consists of three parts. The first part is a set of *propositional variables* or *atomic sentences* (i.e. sentences which have no internal parts). These are letters (we will use **p**, **q**, and **r**) that stand for propositions. Roughly speaking, and setting aside some controversies, a proposition is something that, given a situation, is either true or false. We might at first be inclined to identify propositions with sentences in a language such as English. For example, when I am in my office, we say, "The sentence 'Flynn is in his office' is true", and we say the sentence is false when I am someplace else. But there is some reason to think that propositions are more mysterious abstract entities. Sentences can be ambiguous, such as the oft cited "Visiting relatives can be boring", in which case we might be inclined to say that the sentence expresses two propositions. Also, two different sentences can express the same proposition, as in "Matsui hit a home run" and "A home run was hit by Matsui". By "expressing the same proposition" I mean that we know without having to check that if one of these sentences is true, the other is as well, and likewise with falsity. Further, it seems clear that two sentences drawn from different languages can express the same proposition. Consequently, it appears that there is not a one-to-one correspondence between the propositions and the sentences in natural languages, and therefore we hesitate in identifying them.

Loosely speaking, propositions are the "meanings" of sentences. If a sentence has two meanings, we say it expresses two propositions. If two sentences express the same meaning, we say they express the same proposition. For now, we will discreetly slide interesting questions about the nature of propositions under the rug. Here's all we know about them: propositions are expressed by sentences, and, given a situation, they are true or false. So we let our propositional variables stand for propositions. For example, we might let **p** stand for the proposition expressed by the sentence "Snow is falling (here, now)". Sometimes this will be true, sometimes false.

The second part of the vocabulary for the propositional calculus is the set of *connectives*. These apply to sentences to form other sentences. For our purposes, we will

use only three of these. (Later on, we will consider other connectives and their relation to these three.) We will have one *one-place connective*: ¬. This is the connective for negation and is read "not". We will also have two *two-place connectives*: **&** (read: "and") which is sometimes called *conjunction* , and **v** (read; "or") which is sometimes called *disjunction*. The distinction between one-place and two-place connectives will become clear in a moment.

The third part of the vocabulary consists of the punctuation marks **(** and **)**.

A definition is something that gives instructions (implicitly or explicitly) how to distinguish the thing being defined from everything else. A *recursive definition* is one that sort of "looks back on itself", much as rules in the MIU system could apply to their own output. The following recursive definition of membership in the language of the propositional calculus, which we will call PL, has some rules of the if-then variety (Rules 2-4) and a basis rule (Rule 1) which gives us initial strings:

Rule 1: **p**, **q**, and **r** are sentences in PL.

Rule 2: If α is a sentence in PL, so is $\neg\alpha$.

Rule 3: If α and β are sentences in PL, so is $(\alpha\&\beta)$.

Rule 4: If α and β are sentences in PL, so is $(\alpha v\beta)$.

Rule 5: Nothing is in PL except as specified in Rules 1 through 4.[16]

This system licenses derivations like the MIU system did. For example:

1.	**p**	[1]
2.	**¬p**	line 1, [2]
3.	**q**	[1]
4.	**(¬p&q)**	lines 2,3 [3]
5.	**(pv(¬p&q))**	lines 1,4 [4]

There are a few differences between the PL system and the MIU system. For one, the rules for PL sometimes take as input two sentences, and therefore it will in general not be the case that a particular line in the derivation is licensed by a rule and the immediately preceding line. This calls for a change in our bookkeeping system. Next to each line, we now write the numbers of the all the lines, which serve as input and the number of the relevant rule in square brackets. Otherwise things are very much the same as before. Rules are unordered and may apply anytime their structural descriptions are met, except now the structural descriptions are stated in terms of a feature of derivations that is (as yet) implicit, namely that *every* legal line in a derivation is a sentence in PL. In

[16] We include Rule 5 here just to be explicit that rules 1 through 4 exhaust PL, but from now on we'll take this clause, which is sometimes called the "restriction" for granted. α and β are of course variables in the metalanguage, here ranging over sentences in PL.

the above example, lines 1 through 4 is a demonstration that **(¬p&q)** is a sentence, and this fact is exploited by step 5.

One principle worth mentioning here, even though it may seem completely obvious, is what I will call The Formality Constraint. This principle says that

ONE CAN ONLY DO WHAT THE RULES SAY.

Obvious as it may seem, this sometimes causes problems, especially in this case for people who have worked with some version of PL before. For example, some might be tempted to write line 4 in the above derivation as

¬p&q

but this, as President Nixon used to say in another context, would be wrong. Rule 3 says one must include parentheses, and so, one must. There's a good reason for this feature of the system (which we'll see a bit later on) but for now, take the Formality Constraint to heart. One corollary of the Formality Constraint is important to notice. If you want a system to do something, you need a rule, or at any rate some sort of a licensing procedure, that permits it. Another way to put this is that every step in a derivation must be licensed by an explicit statement in the recognizing system.

Exercise 12

One reason the Formality Constraint is important is this: If you are trying to write a system that will generate a language, if you don't observe the Formality Constraint you likely won't be able to tell if what you propose really does what you think it does, and this might set you off on wild goose chases that last hours or even decades. It is therefore wise to try to keep the Formality Constraint in mind. Write out the Formality Constraint fifty times. (Don't hand this in.) Have a close personal friend tattoo the Formality Constraint on the inside of your forearm.

Exercise 13

PL has infinitely many sentences in it. Write out a demonstration of this.

Exercise 14

Give derivations of the following sentences:

a. **(p&(qv¬r))**
b. **¬(¬¬qvq)**
c. **(r&(q&(p&(r&p))))**
d. **((pvq)&¬(qvr))**

Exercise 15

Parentheses, as you may have noticed, are introduced in pairs. For the sentence (c) in exercise 14, connect with an arc each parenthesis with its "sibling" so that you get a picture sort of like a set of kitchen bowls. (This may seem trivial, but we will soon use this property of PL to demonstrate something more interesting.)

Exercise 16

It can happen that two distinct derivations (distinct in the sense that the rules are applied in a different order) can nevertheless be derivations of the same sentence. Give a derivation of (d) in exercise 14 that is distinct from the one you gave in answering that question.

Here's a bit more about the interpretation of PL. We don't know much about our propositions, but we do know this: given a situation, each one of them is either true or false. We have only three propositional variables, but suppose that I tell you that in a particular situation, **p** and **q** are true and **r** is false. Given the "pronunciations" of the connectives I mentioned earlier, you might then suspect that in that situation, among other things, **¬p** is false, **¬r** is true, **(p&q)** is true, **¬(qvr)** is false, etc. We want to write explicit rules that specify these interpretations.

Our first rule of interpretation will say of our atomic sentences, i.e. the ones that have no internal parts on our analysis so far, that they can be either true or false. Adopting a useful piece of terminology, we will say that each of our atomic sentences *denotes* a proposition, and since, given a situation, propositions are either true or false, each atomic sentence will denote either The True (**T**) or The False (**F**). (This is actually a fairly controversial thing to say, but we set this aside for now.) These are called "truth values". For example, suppose our atomic sentence **p** denoted the proposition, which is also expressed by, the English sentence "It is snowing outside (here, now)." To find out if whether or not **p** is true (or, has the value **T**), one might look out the window. We don't want to fix the interpretation of **p** once and for all, but we do want to restrict its interpretation to truth or falsity, and likewise for the other atomic sentences.

We'll then write rules of interpretation that will fix the interpretations of all of the complex sentences once the interpretation of the atomic constituents is known. We'll do this by linking up each rule in the recursive definition with a rule of interpretation. This perhaps sounds a bit more complicated than it is. I'm confident that after you see the rules and work through some examples, all of this will seem easy.

To emphasize the connection between the rules of the syntax and the interpretation, I've reproduced here the rules from above, and added a rule of interpretation (marked with an "a", for each).

Rule 1: **p**, **q**, and **r** are sentences in PL.

Rule 1a: The atomic sentences denote either **T** or **F.**

Rule 2: If α is a sentence in PL, so is $\neg\alpha$.

Rule 2a: If α denotes **T**, then $\neg\alpha$ denotes **F**. If α denotes **F**, then $\neg\alpha$ denotes T.

Rule 3: If α and β are sentences in PL, so is $(\alpha\&\beta)$.

Rule 3a: If α denotes **T** and β denotes **T**, then $(\alpha\&\beta)$ denotes **T**. Otherwise $(\alpha\&\beta)$ denotes **F**.

Rule 4: If α and β are sentences in PL, so is $(\alpha v\beta)$.

Rule 4a: If α denotes **F** and β denotes **F**, then $(\alpha v\beta)$ denotes **F**. Otherwise $(\alpha v\beta)$ denotes **T**.

These interpretations correspond rather well with our English readings of these connectives. A true sentence prefixed by "not" becomes a false one, and vice versa. A sentence formed by connecting two true sentences with "and" will be true, and false if either part (sometimes called a *conjunct*) is false. A sentence formed by connecting two false sentences with "or" will be false, and true if either part (sometimes called a *disjunct*) is true. (This corresponds to the so-called "inclusive" sense of the English "or", on which the compound sentence is true in case either or *both* of the disjuncts is true.)[17]

Exercise 17

Assuming that **p** is true, **q** is false, and **r** is true, compute the truth values for each of the sentences in exercise 14.

The parentheses play a role in keeping PL unambiguous, where "ambiguity" in this context would be an instance in which fixing the interpretation of the atomic sentences would fail to fix a unique interpretation of the complex sentence. They play a similar role in arithmetic. In the absence of often used disambiguating conventions, a statement like
"5 + 3 x 2" would be ambiguous between one reading on which the value is 16, and the other on which the value is 11. We can disambiguate the string by inserting parentheses around the operation to be performed first, e.g. "(5 + 3) x 2".

Exercise 18

[17] The "exclusive" sense of "or" is said to make a compound sentence false when both disjuncts are true. It's not completely clear that English has this sense of "or" but you can get a feel for it by pondering the standard interpretation of the sentence "You can write a final paper or you can take a final exam."

Consider the sequence **p&¬pvq**. There are three ways of inserting parentheses in this sequence in order to make it well formed in PL. What are they?

Exercise 19

Give the derivations that correspond to each of the sentences you gave in your answer to exercise 18.

Exercise 20

Would the sequence given in exercise 18 be ambiguous, given the rules of interpretation above (and overlooking the parentheses, of course)? By "ambiguous" here I mean is it possible for the string to denote both **T** and **F** in a given situation[18]? Justify your answer.

We'll turn now to developing more tools for describing the syntax of languages. Before we do that, though, let's again look ahead to glimpse the relevance of what we have just seen for the analysis of natural languages. The example of PL shows us that the nature of a derivation can change from system to system. Though every line of a derivation in PL has to be justified by a rule and what came before, this justification can involve any previous line, rather than just the immediately preceding line as in the MIU system. Furthermore, by fixing the interpretation of the "atoms" **p**, **q**, and **r**, we fix the interpretation of all sentences recognized by the system. Notice that, even though there are infinitely many well-formed sentences in PL, once you find out the truth values of the atomic sentences you can determine the truth value of any of the sentences. This is analogous to natural languages like English, in that once you know the rules of the language and learn the meanings of the words, you can compute the meaning of any sentence. The situation in natural languages is surely more complicated, since the languages are more complex and the relevant sense of the notion of "meaning" is perhaps not so clear. But we can now see to a first approximation how it is possible for people to understand sentences they have never heard before. Once you know the meaning of the words, you can use the rules of the language to compute the meaning of any sentence in that language. This is one very strong reason to believe that when people acquire languages (and I'm thinking here of first languages) what they acquire is a rule system. We will return to this issue at some length later on.

---

[18] When we say a sentence is “ambiguous” we usually mean that it has more than one meaning. But this is just about the same thing as saying that the string can be regarded as both true and false in a particular situation, depending on what “reading” one focuses on. For example, take the sentence

> Mary didn’t go to Carleton because of the weather in the wintertime.

It’s easy to see that this is ambiguous by considering two possible elaborations:

> reading A: She went to Pomona instead.
> reading B: She went to Carleton because of its Linguistics Department.

But it’s also easy to imagine a situation where reading A is false and reading B is true, and the citation of such a situation would be taken as evidence that the sentence is in fact ambiguous.

Another difference between PL and MIU is that the vocabulary of PL is differentiated, in the sense that different parts of the vocabulary play different roles in the language. The connectives can be thought of as "bonding agents" for sentences both simple and complex, almost like a chemical bond. The parentheses, on the other hand, a devices that encode the derivational history of the string, in a way that will become clear in the next section.

1.4.2 Trees

If you think about the results of exercises 18 through 20, and compare these with the issue raised in exercise 16, it becomes clear that the order of application of some rules in some cases is relevant to the interpretation of the sentence, while the order of application in other cases is irrelevant. What the parentheses do is encode just those aspects of the *derivational history* of the sentence that may make a difference in the sentence's interpretation. Consider, for example, a sentence like

(¬(p&q)vr).

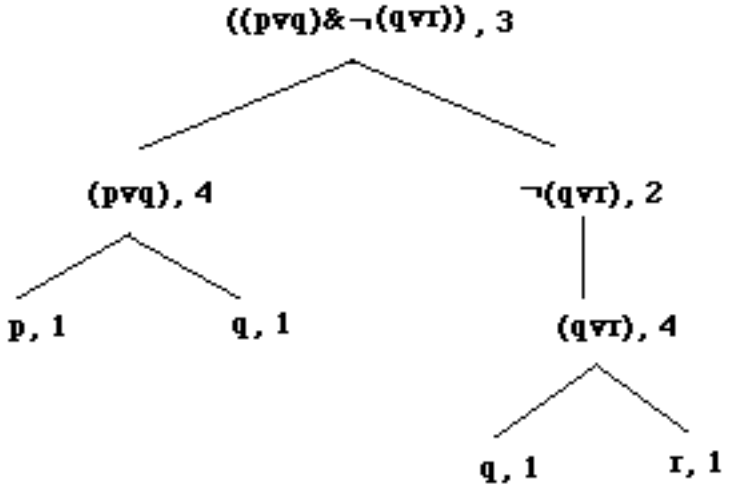
Just looking at this, I can see the relative order in which some rules had to be applied to derive this sentence, but the relative order of application of other rules is impossible to fix with certainty. For example, I know that rule 2 (the rule adding ¬) had to apply before rule 4 (the rule introducing **v**), and this has an effect on the interpretation of the sentence. On the other hand, I cannot tell whether or not **p** (or even **(p&q)**) was introduced before **r**, but this will not make any difference in the sentence's interpretation. What's crucial, as you can probably see after doing the exercises, is the order in which the connectives get "inserted". In the above example, the order has to be **&, ¬, v**.

It's convenient to have a notation that will encode just those aspects of the derivational history that are potentially relevant to the sentence's interpretation, and linguists and philosophers have developed several of these. Let's use sentence (d) from exercise 14 (which is repeated here) as an example in seeing how these notations work.

(14) d. **((pvq)&¬(qvr))**

One kind of representation is called an *analysis tree*. The analysis tree for (14d) appears in (21):

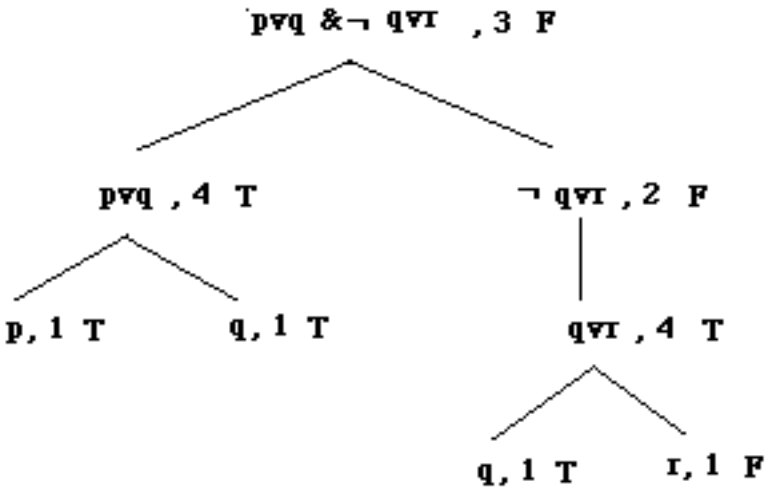
(21)



We borrow some terminology from the arboretum in talking about such (inverted) trees. The lines in the tree are called *branches* and the place where a branch comes together with another (or where it ends) is called a *node*. In the diagram above, I've placed next to each node the number of the rule, which licenses that node. These numbers are not, strictly speaking, a part of the tree. They are there only to help you check to see whether I've followed the rules correctly. The rules themselves have to be reinterpreted in a straightforward way. We now say that a node is permitted if it follows from the nodes below it on the tree by one or another of the rules.

Analysis trees are constructed "from the bottom up". If you compare this tree with the derivation you gave for this string in exercise 12, you'll see that we've lost a bit of information, but the lost information isn't relevant to the interpretation of the sentence. For example, we cannot tell from looking at the tree whether the node **(pvq)** was constructed before or after the node **¬(qvr)**, but this is irrelevant. What is relevant is that, for example, on the right-hand branch rule 4 was applied before rule 2, and this information is easily recoverable by looking at the tree. In a sense, trees correspond to blueprints for a construction project, as opposed to step-by-step instructions. If you're building a table, what's relevant is that you have four legs and a top, but the order in which you attach the legs to the top isn't relevant.

You may have noticed that the tree in (21) is redundant. The information that rule 4 was applied before rule 2 on the right-hand branch is represented in two places. It appears in the tree itself, and also in array of parentheses in the topmost node. Now that we have analysis trees, we can eliminate the parentheses without introducing any ambiguity. To do this, we'd go back to the rules for PL and take out all the parentheses. The tree for (14d) would then look like the one in (22).
(22)



I've taken out the parentheses here, but I've added something else, namely, a possible interpretation for the sentence. I assumed (arbitrarily) that **p** and **q** are both true, and **r** false. There are, of course, other possible interpretations, but once I fix the analysis tree and the assignment of truth values to atomic sentences, the truth value of the topmost sentence is determined. Every (sentence, analysis tree) pair will be unambiguous.

If I take the structure in (22) and erase all but the topmost node, the sentence is ambiguous. Now we can give a first approximation of the notion of *structural ambiguity*. A sentence is structurally ambiguous if it has more than one analysis tree.

Exercise 21

Draw the analysis tree for one "other reading" of **pvq&¬qvr**.

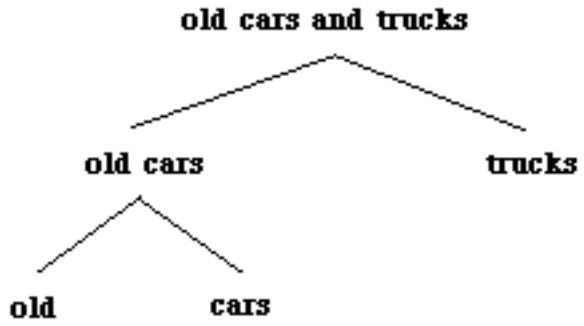
We don't have all the equipment to describe this fully yet, but this situation is one that is frequently encountered in natural languages. Consider for example the sentence in (23).

(23) Old cars and trucks must be inspected by the police.

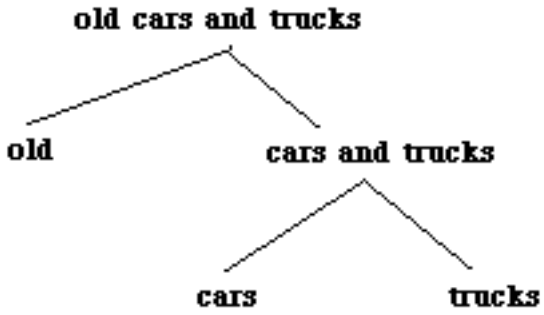
If I own a brand new pickup, I might wonder whether or not this directive applies to me. The reason for this is that the phrase "old cars and trucks" is structurally ambiguous, which is to say that it has both the analysis trees in (24).

(24)

a.



b.



I've suppressed information here (such as the categories of these items and how the word "and" gets into the tree) that we eventually would want to supply, but even so it should be easy for you to see how we will explain the structural ambiguity of this phrase. As you might suspect, it's on reading (a) that I have to get my new pickup inspected, while on reading (b) it's exempt.

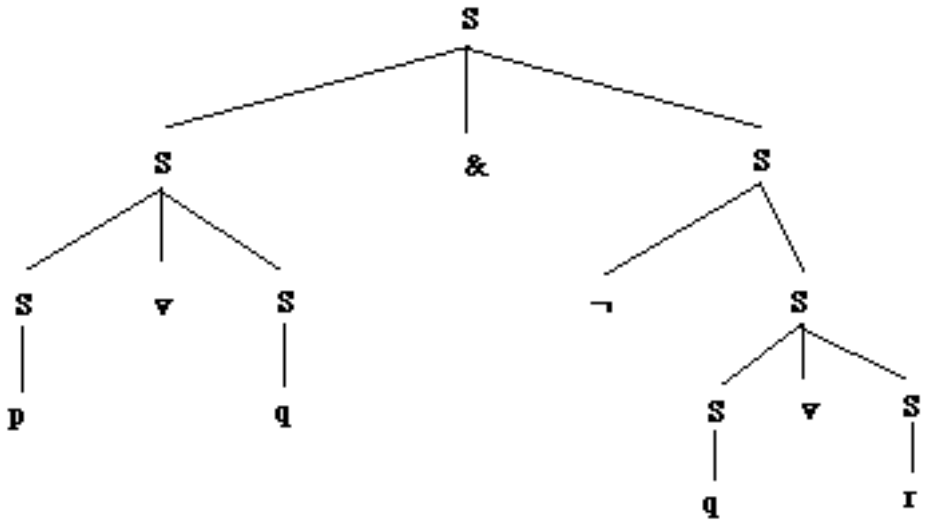
Exercise 22

Give all the analysis trees (without parentheses) that can be associated with the sequence in exercise 18.

Pondering the analysis trees, we might notice that there is one piece of information about this group of nodes that we could make explicit, namely, that each node in the tree is a sentence. (It's perhaps unclear now *why* we might want to make this explicit, since the way our system is set up, every node in a tree will be a sentence, so there is no harm is suppressing this fact in our representations. However, later on we will see that there is often good reason to explicitly represent this sort of categorial information.) A very popular kind of tree that gives a straightforward picture of this "is

a" relation is called a *phrase structure tree* or a *phrase marker*. Consider then the phrase marker for (14d):

(25)



The tree in (25) gives all the information that the one in (22) does, plus it explicitly represents the fact that substrings like **pvq** are sentences. Before we discuss this, let's introduce some helpful terminology. These terms can be defined quite precisely[19] but if we agree to orient our trees from the topmost S downwards on the page, we can get by with the following informal definitions, which will do quite well for our purposes. Nodes and branches are defined as before.

dominance: If from a node A one can move continuously downward (i.e. never turning upward) to reach a node B, then we say that A *dominates* B.

immediate dominance: If a node A dominates a node B, and there is no node C such that A dominates C and C dominates B, then we say that A *immediately dominates* B.

sisterhood: If two or more nodes $A_1,...,A_n$ are immediately dominated by the same node B, then we say that $A_1,...,A_n$ are *sisters*.

daughterhood: If a node A is immediately dominated by a node B, then we say that A is the *daughter* of B.

root: The *root* of the tree is the topmost node.

leaf: The *leaves* of the tree are the bottommost nodes.

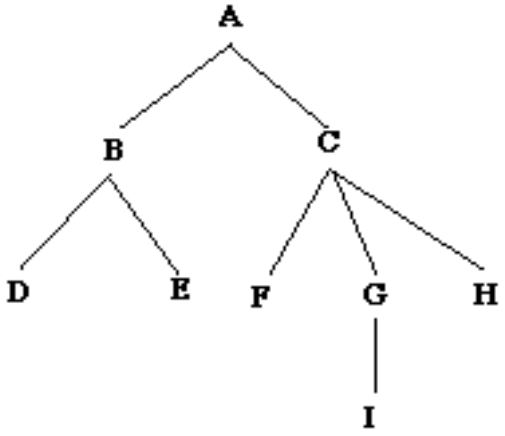
[19]For details, see one or another of the mathematical linguistics textbooks such as Wall 1972.

There's nothing intrinsically special about these definitions. They simply make trees easier to talk about. We can (and will) invent other terms to describe relations between nodes as we need them. Here's some practice with these terms.

Exercise 23

For the following tree, list

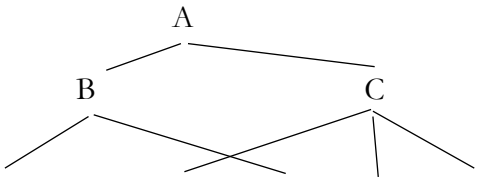
a. the root
b. the leaves
c. the nodes that C dominates
d. the nodes that C immediately dominates
e. the sisters of B
f. the daughters of B
g. the sisters of G



Exercise 24

If a node **A** is a sister of a node **B**, is it always the case that any node **C** that dominates **A** will also dominate **B**?

Let's also agree on something else, which might seem obvious but perhaps is worth making explicit: lines linking nodes to nodes never cross. So, representations like that below we'll say are illegal.



D E F G H
|
I

So, for example, if the mother of a node A precedes the mother of a node B, then A precedes B. We should perhaps stress here that we don't mean to say the crossing lines are impossible in some logical sense, only that, for our purposes, we're going to disallow them until we have some evidence that we need them.

With this terminology in hand, let's return to our phrase marker (25). The parts of the sentence generated appear at the leaves of the tree. It's easy to see how this phrase marker encodes the "is a" relation. A node together with all of its sisters *is a* whatever the label is on the immediately dominating node. For example, the sequence **¬**, **q**, **v**, **r** is an S. The sequence **&**, **¬** isn't anything at all. Here's an important notion: each node in the tree determines a *constituent*, or, as we might say, a recognizable chunk of stuff. More formally,

constituent: All nodes dominated by some node **A** taken together form a *constituent (of type A)*.

In the tree of exercise 21, **F**, **G**, and **H** (and everything they dominate) form a constituent of type **C**. **F**, **G**, and **I** do *not* form a constituent because there is no node that dominates them and only them. (**C** also dominates **H**.)

Exercise 25

Recalling the other sentences from exercise 14 (which are repeated here), give phrase markers for their "de-parenthesized" versions along the lines of (25).

a. **(p&(qv¬r))**
b. **¬(¬¬qvq)**
c. **(r&(q&(p&(r&p))))**

We've seen that trees encode various aspects of a sentence's derivational history, i.e. it displays which rules were applied when in the construction of the string, just like parentheses do in our original version of PL. We looked at two kinds of trees. Analysis trees are constructed "from the bottom up", the node labels themselves are strings in the language, and categorial information is usually suppressed. Phrase structure trees, or phrase markers, are constructed "from the top down", and node labels usually are indications of the category membership of the dominated material. We will mostly deal with phrase markers from here on out, but we should be aware of the option of analysis trees in case we find they would be useful. Both kinds of trees display the structure of strings. A given string might be associated with two or more structures by a given grammar, in which case we say that the string is structurally ambiguous.

1.4.3 Phrase Structure Rules

Though the phrase markers we've been working with accurately display the results of applying the rules for PL last given on p. 19, it turns out to be useful to have a special format for rules that generate trees. While our analysis trees followed a derivation in a "bottom-up" fashion, *phrase structure rules* generate trees "top-down". In this method, one starts with the topmost node, and works down towards the smallest parts of the structure. Collections of phrase structure rules are called *phrase structure (PS) grammars*. Here's a PS grammar for PL:

(26) A PS Grammar for PL (Version 1)

1. **S -> ¬ S**
2. **S -> S & S**
3. **S -> S v S**
4. **S -> p**
5. **S -> q**
6. **S -> r**

(I've numbered the rules for convenience. They are not to be regarded as ordered in any way.) These rules license the construction of phrase markers in the following way: By convention we begin with the symbol **S**.

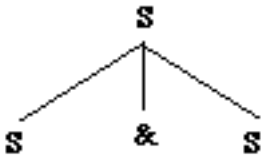
(27) **S**

We then find any rule which has an **S** on the left side of the arrow and apply that rule by writing the symbols on the right side of the arrow underneath the **S** and connect each of these symbols to the **S** with lines. This is sometimes called "expanding" or "rewriting" a node. For example, if we apply rule 5, we get:

(28)

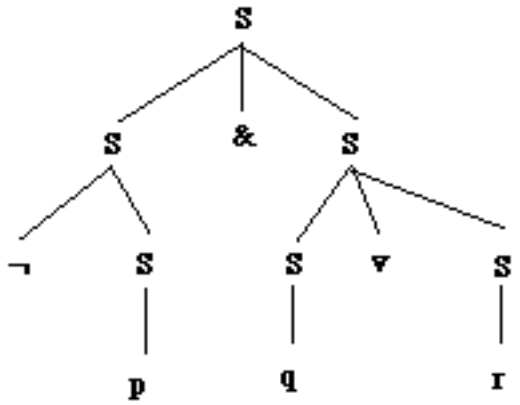
This tree "says" that **q** is an **S** (i.e. a sentence). Since **q** does not appear on the left side of any rule (these symbols are sometimes called *terminals*), this phrase marker cannot be built any further. We say that it is *terminated*. Suppose instead we had chosen rule 2:

(29)



Now we can expand the other two **S** nodes, again by choosing any rule that has an **S** on the left side of the arrow, and continue this until the bottom nodes can no longer be expanded by any rule. The following completion of the tree should be easy to follow:

(30)



In our original PL system, this would be the sentence **(¬p&(qvr))**. Both say more or less the same thing. In fact, we could make them exactly equivalent by adding node labels to the parentheses. When linguists do this, its common practice to use square brackets instead of parentheses. So, an alternative representation of the tree in (30) is (31):

(31) **[s [s ¬ [s p]] & [s [s q] v [s r]]]**

Which representation one chooses, trees or labeled bracketing, usually depends on what will be easiest to read. In this case, you probably find the tree easier to absorb, but if I was keen to emphasize (for some reason) that **r** in (30) is "three S's down", I might choose a labeled bracket notation, and leave irrelevant parts of it out to help you focus on this fact better:

(32) **[s ... [s ... [s r]]]**

It's very helpful to get very good at translating from the tree notation to labeled brackets and back again.

Exercise 26

Draw three trees generated by the grammar in (26) that you have never seen before. Give the equivalent representations in labeled bracketing.

We've now modified our metalanguage so that we explicitly represent the fact that in PL strings like **p**, **¬r**, and **q&p** are sentences. Our loyal and hard-working connectives might be feeling a bit neglected, so in an egalitarian spirit (and with forthcoming ulterior motives up our sleeve), let's assign them to categories as well. We could of course

choose any labels for these categories we like, so we'll choose ones that (like **S**) abbreviate our English names for these categories: **1CON** and **2CON**.

(33) A PS Grammar for PL (version 2)

S -> 1CON S
S -> S 2CON S

1CON -> ¬
2CON -> &
2CON -> v
S -> p
S -> q
S -> r

Exercise 27

Give trees generated by the grammar in (33) that correspond to those you drew in exercise 25.

In the grammar in (33) I've separated the rules into two groups in order to draw your attention to a distinction we might make between types of rules in this system. To characterize this distinction, think of how you might describe PL to someone. You might say, "There are three kinds of categories in PL, sentences, one-place connectives, and two-place connectives. As for the architecture of complex sentences, one-place connectives precede the sentences they go with, and two-place connectives appear between the two sentences they go with." Pursuing the analogy with natural languages we introduced earlier, we might say that there are three kinds of "words" in PL, atomic sentences and the two kinds of connectives. The rules in the second group in (33) serve to introduce the "words" into trees. The two rules in the first group specify the structure of the complex sentences.

The list of words in a language is sometimes called the *lexicon*, and we can call the rules in the second group *lexical insertion rules*, since they insert lexical items into trees. You might have the intuition that modifications of the lexical insertion rules wouldn't change the fundamental structure of the language very much. For example, adding a new atomic sentence with a rule **S -> t** is rather like an English speaker learning a new word. Technically, of course, this does change the language generated, but the new one overlaps with the old so much that we still might regard the two languages as being in some basic sense the same. Changing or adding to the first group of rules, on the other hand, would seem to change the language in a more radical fashion.

We can recognize this distinction while at the same time streamlining our grammar of PL by making a third change in the metalanguage. We separate the rules into two *components*. The *Lexical Component* consists of rules, which list the terminal symbols and their categories. It is rather like (and sometimes is called) a dictionary. The *Phrase Structure Component* consists of rules that specify the structure of complex

sentences. These rules most often have at least two category (or *nonterminal*) on the right side of the arrow.

(34) A PS Grammar for PL (Version 3)

Phrase Structure Component:

S -> 1CON S
S -> S 2CON S

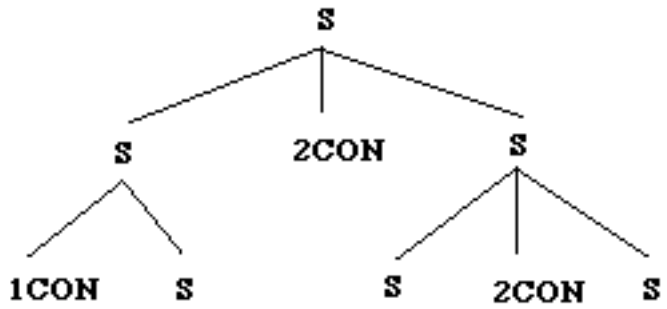
Lexical Component:

S: p,q,r
1CON: ¬
2CON: &,v

We read the Lexicon as follows: The symbol to the left of the ":" is the category of each of the items to the right. When constructing a tree, we are allowed to insert any item of given category underneath the node of the tree, which matches the category of the given item.

This change in the metalanguage doesn't change PL at all. It only alters our view of how derivations proceed. They now have two distinguishable parts. In the first part, the PS rules generate a tree such as that in (35).

(35)



Then the lexical component inserts appropriate terminal symbols underneath what we might call the *preterminal* nodes of the tree (i.e. those nodes which immediately dominate lexical items).

Our metalanguage now gives us an easy way of distinguishing what are surely two very different processes in natural languages. Learning (or forgetting) a word is now characterizable as an addition to (or deletion from) the lexicon.

Exercise 28

Suppose I tell you that ≡ is a two-place connective, ⊗ is a one-place connective, and **s** is an atomic sentence in PL. Which of the following is well formed according to the grammar in (34)? For those that are, give a tree generated by that grammar, assigns a structure.

a. **p≡q&⊗s**
b. **p⊗s≡q**
c. **≡p&s⊗r**
d. **⊗⊗p≡s≡r**

1.4.4 The Fable of the Planet of the Propositional Logic Speakers

To get a picture of the second kind of rule modification in natural languages, let's indulge ourselves with a bit of fancy.

On a planet far far away in another galaxy, there lives a group of creatures that speak languages not unlike our old friend PL. I won't describe their physical appearance and you are invited to imagine them how you wish. On this planet, there are many different countries and there are many different languages, with wildly diverging sound systems and orthographic conventions. We'll actually simplify matters somewhat by giving sentences in these languages that are to be read from left to right, as in English. (Some groups on the planet write their sentences from right to left or top to bottom.)

One group speaks a language called Camestres which happens to be exactly like PL, including the additions made in exercise 28. Camestranians obviously don't talk about much, and the extent of their thought processes is an open question.

Not far from the Camestres speakers lives a group that speaks Bocardo. The PS component for Bocardo is the same as Camestres, but the lexicon looks quite different. Here's some translations:

(36) Glosses between Camestres and Bocardo:

Camestres	Bocardo
p	**Π**
q	**Θ**
r	**Ρ**
s	**Σ**
¬	**∠**
⊗	**©**
&	**√**
v	**↓**
≡	**™**

Even though their orthographies are quite different, the pronunciations of the languages are identical, and this has tended to intertwine the two economies, with all the usual resulting frictions.

Exercise 29

Translate the following sentences into Bocardo.

a. **p&qv¬r**
b. **pvq&¬qvr**
c. **⊗⊗p≡s≡r**

The relation between Camestres and Bocardo is rather straightforward (and hence rather uninteresting). For example, both languages have ambiguous sentences, which both groups regard as a valuable feature of the languages. But the relation between both of these languages on the one hand and Fresison on the other is more intriguing. Fresison has a lexicon and orthography exactly like Camestres, but instead of the rule **S -> S 2CON S**, Fresison has **S -> 2CON S S**. In other words, instead of the two-place connectives appearing between the two sentences they connect, in Fresison all the connectives appear before the sentences they apply to.

Exercise 30

Translate the sentences in exercise 29 into Fresison, and give the trees for these sentences. There may not be a unique translation, since as you'll recall, sentences in Camestres are sometimes ambiguous. In Fresison, it turns out, no sentences are

ambiguous (the Fresisonians take this as a particular source of pride), and consequently it is often the case that a perfectly faithful translation is impossible. So for this exercise, pick one or another of the readings of the ambiguous strings in Camestres for translation, and indicate which reading you've chosen by drawing a tree for the Camestranian sentence.

Bramantip, like Fresison, has a vocabulary, phonology, and orthography identical to Camestres, but it's PS Component looks like this:

The PS Component for Bramantip:

S -> S 1CON
S -> S S 2CON

Exercise 31

Translate the sentence in exercise 29 into Bramantip. (Bramantip, like Fresison, also has no ambiguous sentences, so the same instructions from exercise 27 apply here. While doing this exercise you may see why the Bramantipites think the Fresisonians do everything backwards, and vice versa.)

Felapton is interesting. It is like Fresison in syntax, and it is written like Fresison, but its phonology is quite different. (Korean and Japanese have nearly identical structures but very different phonologies.) Here's some pronunciations:

p: 'nip'
q: 'nop'
r: 'nope'
¬: 'heep'
&: 'hoop'
v: 'hop'

Exercise 32

Translate the following utterance in Felapton into Camestres:

hoop hop heep nip nop nope

Were we to venture further, we would encounter apparent limitless diversity, what with all the varying orthographies, pronunciations, modifications of vocabulary and variations on the PS rules. But from our point of view, we can see that beneath this rich diversity there is a fundamental sameness. All of these languages are really variations on

a single theme. This is something that the inhabitants of the planet might find worth knowing.

Exercise 33

During your exploration of the Planet of the Propositional Logic Speakers, you discover a heretofore unknown language. Describe it.

Bibliography

Botha, Rudolf P. (1989) *Challenging Chomsky.* Oxford: Basil Blackwell.

Chomsky, Noam (1957) *Syntactic Structures.* The Hague: Mouton.

Chomsky, Noam (1959) "A Review of B.F. Skinner's *Verbal Behavior,*" *Language* 35.1.26-58. Reprinted in Jerry A. Fodor and Jerrold J. Katz (eds.) (1964) *The Structure of Language: Readings in the Philosophy of Language.* Englewood Cliffs, NJ: Prentice-Hall, Inc.

Chomsky, Noam (1965) *Aspects of the Theory of Syntax.* Cambridge, MA: MIT Press.

Chomsky, Noam (1995) *The Minimalist Program.* Cambridge, MA: MIT Press.

Delong, Howard (1971) *A Profile of Mathematical Logic.* Reading, Mass.: Addison-Wesley Publishing Company.

Frazier, Lyn (1978) *On Comprehending Sentences: Syntactic Parsing Strategies.* University of Connecticut doctoral dissertation.

Frege, Gottlob (1879) *Begriffsschrift.* (Chapter 1 appears in English translation in Peter Geach and Max Black (eds.) *Translations from the Philosophical Writings of Gottlob Frege.* Oxford: Basil Blackwell, 1970.)

Gleitman, Lila R. (1981) "Maturational Determinants of Language Growth," *Cognition* 10: 103-114.

Hofstadter, Douglas R. (1979) *Gödel, Escher, Bach: An Eternal Golden Braid.* New York: Vintage Books.

Langacker, Ronald W. (1987) *Foundations of Cognitive Grammar, vol. 1: Theoretical Prerequisites.* Stanford: Stanford University Press.

Lewis, Harry R. and Christos H. Papadimitriou (1981) *Elements of the Theory of Computation.* Englewood Cliffs, New Jersey: Prentice-Hall, Inc.

Pinker, Steven (1994) *The Language Instinct.* New York: William Morrow and Company, Inc.

Prior, A. N. (1962) *Formal Logic.* Oxford: Oxford University Press.

Skinner, B.F. (1957) *Verbal Behavior.* New York: Appleton-Century-Crofts, Inc.

Smith, Neil (1989) *The Twitter Machine: Reflections on Language.* Oxford: Basil Blackwell.

Wall, Robert (1972) *Introduction to Mathematical Linguistics.* Englewood Cliffs, New Jersey: Prentice-Hall, Inc.

Whitehead, Alfred North and Bertrand Russell (1910-1913) *Principia Mathematica*. Cambridge: Cambridge University Press.

Index