

Transitive Verbs, Continued

LING 553

September 17, 2008

1. RETURN OF THE LAMBDA CALCULUS

Recall from page 8:

ψ is that function from the (set of) positive integers to the (set of) positive integers which, given a positive integer, returns its successor.

can be written:

that function...	from positive integers	to	positive integers, which,
given a	positive integer,	returns	its successor
[λx	: $x \in \mathbb{N}$.	$x + 1$]

That wasn't just needless notation; we can use it now to simplify

[[**snores**]] = that function from entities to truth values such that, given an entity, it returns TRUE if and only if that entity snores

[[**sees**]] = that function from entities to [functions from entities to truth values] such that:
given: an entity (call it a),
it returns: that function from entities to truth values such that:
given: an entity (call it b),
it returns: TRUE if and only if entity b sees entity a .

into

- (1) [[**snores**]] = $[\lambda x : x \in D_e . x \text{ snores}]$
- (2) [[**sees**]] = $[\lambda x : x \in D_e . [\lambda y : y \in D_e . y \text{ sees } x]]$

1.1. *Some Important Asides: Notation*

There are some notational variants of this. I won't use them in this class—and you shouldn't use them on homework—but you'll want to recognize them.

- Because the type of the variable is specified, any variable can be used— x , P , $fred$, or whatever. But there are some conventions:
 - x, y, z are usually variables of type e^1
 - P, Q are usually variables of type $\langle e, t \rangle$
 - R is often a variable of type $\langle e, \langle e, t \rangle \rangle$
- Occasionally you'll see $[\lambda x : x \in D_\sigma . \text{blah}]$ abbreviated as $[\lambda x \in D_\sigma . \text{blah}]$, or even as $[\lambda x_\sigma . \text{blah}]$. These are informal, but fortunately unambiguous. In general, a type subscript on a variable may be used to indicate that the variable is of the given type.
- Brackets are more or less optional; they can be left out, or more can be added for clarity. So (2) may be written $[\lambda x : x \in D_e . \lambda y : y \in D_e . y \text{ sees } x]$, or even (following the above note) $[\lambda x_e . \lambda y_e . y \text{ sees } x]$. Sometimes, when the variables are being used in their standard ways as described above, or when preceded by a note saying, e.g., “The letters $x, y, z \dots$ will be used as individual variables of type e ” (Romero 2005), the domain restriction may be left off entirely: $[\lambda x . \lambda y . y \text{ sees } x]$.
- The truth conditions that (1) and (2) yield (“ x snores”; “ y sees x ”) are written out in the metalanguage. The more mathematically inclined a semanticist is, the more likely they are to use “predicate logic”, in which predicates are written as functions, and truth conditions are left as functions with arguments. For instance, instead of

$$[[\text{snores}]] = [\lambda x \in D_e . x \text{ snores}]$$

they may use

$$[[\text{snores}]] = [\lambda x \in D_e . \text{snores}'(x)] = \text{snores}'$$

$$[[\text{Julie snores}]] = \text{snores}'(j)$$

where the prime mark is used to indicate a function as opposed to a word of the object language (or of the metalanguage).

We're not going to do that here, because writing out “ x snores” is (a) easier to understand and (b) a good reminder that we're working with actual descriptions of the world, and not just abstract mathematical results.

There are other notational things, most of them used only early in the field (e.g., $\hat{\mathbf{a}}$ as a shorthand for $\lambda \mathbf{a}$), but these are the main ones you may see.

¹ Note, though, that in some work from the 1970s, u and v are individuals, and x and y are used for a different type.

2. SYNTAX AND TRANSITIVE VERBS

Above, it was decided that $[[\text{sees}]] = [\lambda x : x \in D_e . [\lambda y : y \in D_e . y \text{ sees } x]]$. But that was based on a particular syntax we assumed. What if we'd assumed a different syntax?

	Syntactic Structure	Meaning for sees
(i)	<pre> S / \ Julie VP / \ sees Charles </pre>	$[\lambda x : x \in D_e . [\lambda y : y \in D_e . y \text{ sees } x]]$
(ii)	<pre> S / \ XP Charles / \ Julie sees </pre>	$[\lambda x : x \in D_e . [\lambda y : y \in D_e . x \text{ sees } y]]$
(iii)	<pre> S / \ Julie sees Charles </pre>	$[\lambda \langle x, y \rangle : x, y \in D_e . x \text{ sees } y]$

We assumed the syntax in (i). If instead we had assumed (ii), the semantics would look much the same—except that the first argument that $[[\text{sees}]]$ takes would be put into subject position.

As for (iii)...well, not many modern syntacticians would draw a tree like that (at least, not syntacticians in the Chomsky tradition). For one thing, it's long been known that any tree with ternary branching can be written as a tree with binary branching (aka a tree in "Chomsky-normal mode").

But from the standpoint of predicate logic, that meaning for **sees**—and thus, that structure—make a lot of sense. There's an intuition that a transitive verb represents a relation between two individuals: **Julie sees Charles** means that a particular relationship holds between Julie and Charles, namely that the former sees the latter. In fact, in predicate logic—the same formalism that prefers to represent "Julie snores" as **snores'(j)**, or **S(j)**, or even just **Sj**—"sees" represents a two-place relation written **R(j,c)** or **Rjc** or **jRc**. Logicians like that sort of thing; and it's not that bad an intuition.

2.1. Why not ternary branching?

So why not represent things as in (iii)?

- Constituency arguments. Syntacticians tell us (i) and not (ii) is the structure because they have a variety of reasons to think that [**sees Charles**] is a unit in a way that [**Julie sees**] is not. Those same arguments suggest that (iii) is wrong, because no special status is accorded to either substring.
- Cross-linguistic facts. (iii) requires a rule that says how to combine $[NP_1 V NP_2]$ —a rule that is specific to those labels, which runs contrary to the attempt to simplify everything to geometry. And because it's specific to that structure, it would only be an interpretation rule in an SVO language like English; in an SOV language like Japanese or Basque, we'd need to have an interpretation rule for $[NP_1 NP_2 V]$. If we group the verb and its object together, then both kinds of language will just need rules for $[V NP_2]$ -in-some-order and $[NP_1 VP]$ -in-some-order.
- ...to say nothing of alternate structures in English.
 - Ditransitive verbs: we'd need yet another rule to handle the quaternary-branching structure in **Julie gave Charles a book**.
 - Topicalization: would **Bagels, I like (but kaiser rolls, I hate)** have to have a different interpretation rule for $[NP_2 NP_1 V]$?

So there are plenty of good reasons not to use (iii). And yet it's so appealing to logicians.

2.2. Currying

Just as a ternary-branching structure can be turned into a binary-branching structure, currying² is a method of taking a function from pairs, and turning it into a function from one element of the pair to functions from the other element of the pair. That is, it'll turn a function of type $\langle \alpha, \beta \rangle, \gamma$ and turn it into a function of either type $\langle \alpha, \langle \beta, \gamma \rangle \rangle$ or $\langle \beta, \langle \alpha, \gamma \rangle \rangle$. (In particular, in this case, from $\langle \langle e, e \rangle, t \rangle$ into $\langle e, \langle e, t \rangle \rangle$.)³ This means that we can formally express a correspondence between a relation on ordered pairs and a nested function.

² Heim & Kratzer use *Schönfinkel* and *Schönfinkelization* on the grounds that Haskell Curry was building on the work of Moses Schönfinkel. Molly Diesing suggested that they should stick with “curry”, on analogy with Stephen Jay Gould's argument about the brontosaurus—the term entered general use before the animal was recognized to be the earlier-named apatosaurus, and one might as well stick with the common, generally used term over the technically-correct-by-precedence term. H&K reject this on the grounds that “We are not sure how general the use of ‘Currying’ is at this time” (p. 41); but it's probably well-accepted enough in computer science circles to justify using it over the technically-correct “Schönfinkel”.

³ Actually, currying will take a function from n -tuples and convert it to a function that takes the elements of the n -tuple one at a time, analogous to turning a flat structure with n branches into a nested binary-branching structure. But we'll only need pairs here.

For instance: suppose $U = \{t, j, b\}$, which are Tony, Julie, and Beatrice; suppose, too, that Tony is the tallest, and Beatrice the shortest. Then there is a two-place function “is-taller-than”:

- $[[\text{is-taller-than}]] = \lambda \langle x, y \rangle \in U \times U . x \text{ is taller than } y$

In this universe, that's $\{\langle t, j \rangle, \langle t, b \rangle, \langle j, b \rangle\}$, or:

$\langle t, t \rangle \rightarrow$	FALSE
$\langle t, j \rangle \rightarrow$	TRUE
$\langle t, b \rangle \rightarrow$	TRUE
$\langle j, t \rangle \rightarrow$	FALSE
$\langle j, j \rangle \rightarrow$	FALSE
$\langle j, b \rangle \rightarrow$	TRUE
$\langle b, t \rangle \rightarrow$	FALSE
$\langle b, j \rangle \rightarrow$	FALSE
$\langle b, b \rangle \rightarrow$	FALSE

Schönfinkelization will give a new function f such that $f(x)(y)$ is true whenever $[[\text{i-t-t}]](\langle x, y \rangle)$ is true. In particular, $f(x) = [\lambda y : y \in U . \text{TRUE iff } [[\text{i-s-t}]](\langle x, y \rangle)]$. In this case:

“Tony is taller than x ”

$$f(t) = \begin{bmatrix} t \rightarrow \text{FALSE} \\ j \rightarrow \text{TRUE} \\ b \rightarrow \text{TRUE} \end{bmatrix}$$

“Julie is taller than x ”

$$f(j) = \begin{bmatrix} t \rightarrow \text{FALSE} \\ j \rightarrow \text{FALSE} \\ b \rightarrow \text{TRUE} \end{bmatrix}$$

“Beatrice is taller than x ”

$$f(b) = \begin{bmatrix} t \rightarrow \text{FALSE} \\ j \rightarrow \text{FALSE} \\ b \rightarrow \text{FALSE} \end{bmatrix}$$

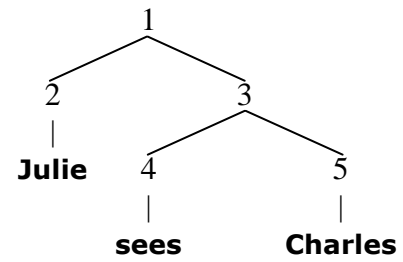
and f , our new $[[\text{is-taller-than}]]$, is

$$\begin{bmatrix} t \rightarrow \begin{bmatrix} t \rightarrow \text{FALSE} \\ j \rightarrow \text{TRUE} \\ b \rightarrow \text{TRUE} \end{bmatrix} \\ j \rightarrow \begin{bmatrix} t \rightarrow \text{FALSE} \\ j \rightarrow \text{FALSE} \\ b \rightarrow \text{TRUE} \end{bmatrix} \\ b \rightarrow \begin{bmatrix} t \rightarrow \text{FALSE} \\ j \rightarrow \text{FALSE} \\ b \rightarrow \text{FALSE} \end{bmatrix} \end{bmatrix}$$

This in particular was **LEFT-TO-RIGHT** currying (so called because we moved through the ordered pairs left to right). Because we want verbs to combine with their objects first, and then their subjects, we'll use **RIGHT-TO-LEFT** currying. That is, instead of having each $f(\alpha)$ represent the “things that α is taller than” function, they'll represent the “things that are taller than α ” function. (Which is how we did it in class; reconstructing that is left as an exercise to the reader.) Of course, note that if syntacticians had told us that the structure to interpret was (ii) instead of (i), then left-to-right currying would have gotten us the meaning we wanted.

3. HOW TO WRITE DERIVATIONS

So we know what interpretation we get for **[[Julie sees Charles]]**, given the tree to the right. But it's important to be able to write out a derivation, i.e. a series of steps that demonstrates how the meaning is derived. We can do that *top-down* or *bottom-up*. Either way, we need a structure (i.e., that thing on the right), and a set of interpretation rules and lexical meanings.



- (3) a. **The Non-Branching Node Rule (NBNR)**
 If we have a node α , and its only daughter is β , then $[[\alpha]] = [[\beta]]$.
- b. **The Rule of Functional Application (FA)**
 If we have a node α , and its daughters are β and γ , where β has type $\langle \sigma, \tau \rangle$ and γ has type σ , then $[[\alpha]] = [[\beta]]([[\gamma]])$.
- c. **The Lexical Meaning Rule (LM)**
 If α is a terminal node, its interpretation $[[\alpha]]$ is its lexical meaning.
- (4) **[[julie]]** = Julie
[[charles]] = Charles
[[sees]] = $[\lambda x : x \in D_e . [\lambda y : y \in D_e . y \text{ sees } x]]$

3.1. Top-Down

A top-down derivation consists of a series of lines, each equivalent to the previous line, each line annotated with the rule that licenses the equivalence. (Note: the order of operations doesn't matter.)

$[[1]]$	$= [[3]]([[2]])$	Function Application (FA)
	$= [[4]]([[5]])$	FA ⁴
	$= [[\text{sees}]]([[5]])$	Non-Branching Node (NBNR)
	$= [[\text{sees}]]([\text{Charles}])$	NBNR $\times 2$
	$= [[\lambda y_e . [\lambda x_e . x \text{ sees } y]]](\text{Charles})$	Lexical Meaning (LM)
	$= [[\lambda y_e . [\lambda x_e . x \text{ sees } y]]](\text{Charles})(\text{Julie})$	LM $\times 2$
	$= [\lambda x_e . x \text{ sees Charles}](\text{Julie})$	λ -conversion
	$= \text{TRUE iff Julie sees Charles}$	λ -conversion

⁴ $[[3]]$ has been replaced by $[[4]]([[5]])$. This line could read simply $[[4]]([[5]])$, but per Dimka's suggestion, brackets have been added to emphasize that whatever $[[4]]([[5]])$ is, it'll take $[[2]]$ as its argument.

3.2. *Bottom-Up*

A bottom-up derivation consists of a series of steps, each of which gives the meaning of a node, each containing a series of equivalent lines, again annotated with the rule that licenses the equivalence.

1.	[[Charles]]	= Charles	LM
2.	[[5]]	= [[Charles]] = Charles	NBNR step 1
3.	[[sees]]	= $[\lambda y_e . [\lambda x_e . x \text{ sees } y]]$	LM
4.	[[4]]	= [[sees]] = $[\lambda y_e . [\lambda x_e . x \text{ sees } y]]$	NBNR step 3
5.	[[3]]	= [[4]] ([[5]]) = $[\lambda y_e . [\lambda x_e . x \text{ sees } y]]$ ([[5]]) = $[\lambda y_e . [\lambda x_e . x \text{ sees } y]]$ (Charles) = $[\lambda x_e . x \text{ shot Charles}]$	FA step 4 step 2 λ -conversion
6.	[[Julie]]	= Julie	LM
7.	[[2]]	= [[Julie]] = Julie	NBNR step 1
8.	[[1]]	= [[3]] ([[2]]) = $[\lambda x_e . x \text{ sees Charles}]$ (Julie) = TRUE iff Julie sees Charles	FA steps 5 and 7 λ -conversion