Semantics 564 **Lecture 13** Oct. 7, 1999

1 Types and sentential connectives

Recall that last time we had the following as our possible types of denotations:

A: a bunch of possible types of denotations

i) elements of D, the set of individuals
ii) elements of {1,0}, the set of truth values
iii) functions from D --> {1,0}

Let's adopt a notation, whereby we use "D" to indicate any set that includes all elements of a given type, and subscript it with a letter or formula that indicates what the particular type in question is. So for i) and ii) we'll have i') and ii'), below:

A: i') D_e
ii') D_t

where "e" stands for *entities*, that is individuals in the real world, so i') is the set of entities, and "t" stands for *truth values*, that is {1, 0}, so ii') is the set of truth values. What's iii') going to be? Let's reword it:

A: iii') the set of functions from D_e to D_t.

Now, what are functions from the set of all x to the set of all y? Simply a set of ordered pairs, the first of whose members is an x and the second of whose members is a y, that is, a bunch of <x,y>. (Note of course that this set of ordered pairs, in order to be a function, must map a particular y to every x, so it's not the same thing as the Cartesian product of the two sets, but rather a (proper) subset of that Cartesian product). In any case, for notational purposes, we'll say that the *type* of the function in iii) and iii') is <e,t>, since its domain (its arguments) are entities and its range (its values) are truth values, and we can notate the set of all such functions as iii"):

iii") $D_{<e,t>}$

More generally, we can make a recursive definition of permissible types in the following way:

1. (i) e and t are semantic types

(ii) If and are semantic types, then < , > is a semantic type

(iii) Nothing else is a semantic type.

(iv) For any two types, ,

< , > is defined to be the type of functions whose arguments (domain) are of type and whose values (range) is of type .

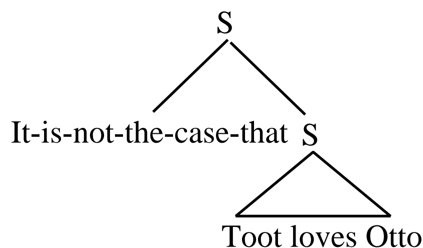
(v) The set of entities is D_e.

(vi) The set of truth values is D_t

(vii)The set of functions of type < , > is $D_{< , >}$.

Ok, now we're ready to proceed. Let's turn to the exercise on sentence connectives on page 23. How do we treat the following:

2. It-is-not-the-case-that [$_S$ Toot loves Otto].



They warn us that we're going to invent a new possible semantic value, i.e. type (presumably for the connective, since we know what the semantic values for Ss are; they're truth values). So what is the type of the connective it-is-not-the-case-that going to be?

Well, we know the type of S, it's a truth value. Truth values are not functions, they're saturated expressions, like entities. If the binary-branching node under the topmost S is going to represent a functional application, then the "word" **it-is-not-the-case-that** must be a function that takes truth values as its arguments, since its sister's type is a truth value. And what will this function give as its value? Looking at the syntactic type of the combination of **it-is-not-the-case-that** and S, we see that it also is an S — and the type of Ss is truth values. So **it-is-not-the-case-that** must be a function that takes truth values as arguments and gives truth values as values, a function of type <t, t>, a type which is allowed in our definition of types above.

3. [[**it-is-not-the-case-that**]] = f: {0,1} to {0,1}

...

And what will we fill in the ellipse with? Let's consider our predicate-logical definition of **it-is-not-the-case-that:**

4. [[ÿP]] = 1 iff [[P]] = 0

So it should be pretty obvious: this function will give the value 1 iff the value of its argument is 0.

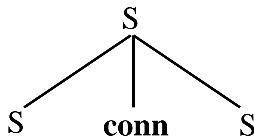
[[**it-is-not-the-case-that**]] = f: {0,1} to {0,1}

For all x D_t, f(x)=1 iff x=0

So we've introduced a new type, $D_{<t,t>}$, and defined a new lexical item, **it-is-not-the-case-that**. We don't need a new rule to interpret the structure of "It is not the case that Toot loves Otto", because our rule for S's works perfectly well (if you ignore the linearity problem).

2 Transitive functions

Let's consider the other sentential connectives, *and* and*or*. Now, the trick with these is that H&K give ternary-branching structures as the representation of these, yet we have to stick with the idea that syntactic combination is functional application.



We know, as above, that the value of the S nodes is a truth value, not a function, so the two S nodes must be arguments. The connective, again, must be the function, and since there's two values immediately syntactically connected to it, this function must take not one truth value, but rather a pair of truth values <x,y>, as its argument. This will be a 2-place function. The pair of truth values it takes will be a member of {0,1}x{0,1}, since either sentence could have either truth value. So these functions will look like this:

5. [[**and**]] = f: {0,1}x{0,1} to {0,1}

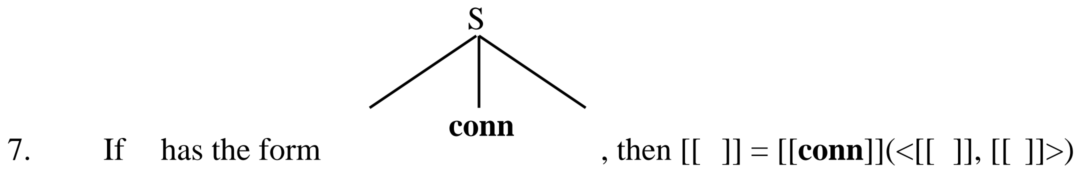
For all <x,y> {0,1}x{0,1}, f(<x,y>)=1 iff x=1 and y=1

(This is the characteristic function of the set {<1,1>}

6. [[**or**]] = f: {0,1}x{0,1} to {0,1}
For all <x,y> {0,1}x{0,1}, f(<x,y>)=1 iff x=1 or y=1

(This is the characteristic function of the set {<1,0>, <1,1>, <0,1> }

And now that we've got our new lexical entries and our new type, we need a structural rule for interpreting the ternary branching structure. It will look like this:



7. If has the form , then [[ ]] = [[**conn**]](<[[ ]], [[ ]]>)

Now what about transitive verbs? Consider what our denotations for them in first order predicate logic were:

8. [[**love**]] {<x,y> D_e x D_e | x loves y}

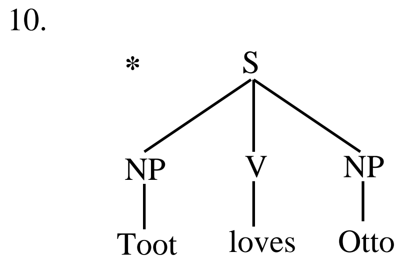
Again, they were sets of ordered pairs. Now, we could just make our function which denotes **love** the characteristic function for this set, just as we did for the sentential connectives above:

9. [[**love**]] = f: D_e x D_e to {0,1}
For all <x,y> D_e x D_e , f(<x,y>)=1 iff x loves y.

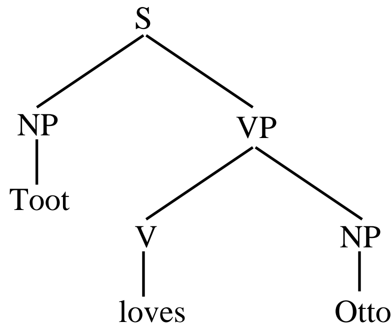
We could do this. But it would be wrong. Why would it be wrong?

It would be wrong because the structure of "Toot loves Otto" isn't the one given in 10, but rather the one given in 11:

10.



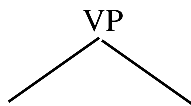
11.



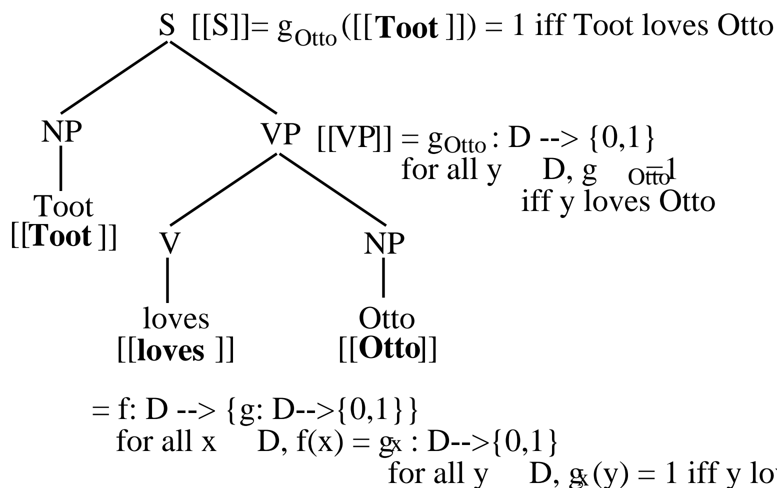
So, if 11. is the structure of "Toot loves Otto", what does the verb "Love" have to do? We know that S has the denotation of a truth value, and that "Toot" and therefore NP has the denotation of an entity, so the denotation of VP has to be a function from entities to truth values. That's not so hard. Now the tricky part: if VP has the denotation of a function from entitites to truth values, and the NP dominating "Otto" has the denotation of an entity, then "love" has to be a *function from individiuals to functions from individuals to truth-values.* That is, it has to take an individual as its argument, and give as its value a function from individuals to truth values -- a particular function. Here it is:

12. [[**loves**]] = f: D_e --> {g: D_e --> {0,1}}
for all x D_e , f(x) = g_x: D_e --> {0,1}
for all y D_e , g_x(y) = 1 iff y loves x

Of course, we also have to give a new rule for interpreting the VP node, since so far we only know how to interpret non-branching VPs. Since all branchingness represents functional application, the new rule will look like this:

13. If has the form  , then [[ ]] = [[ ]]([[ ]])

13.



What's the type of the function for **loves**? It's a function from entities to functions from entities to truth-values, so its < e, <e,t> >. And the set of all such functions is $D_{<e, <e,t>>}$. (This is a permissible type, according to our recursive definition of types, above). However, it's important to realize that this is a 1-place function, whose value is another 1-place function. Yet "love" seems to take two arguments, i.e. in our set-theoretical notation, it denoted a set of pairs of entities, and our first attempt at defining **love** took the charasteristic function of that set of pairs as its denotation. What's the relation between that characteristic function and the 1-place function we've arrived at here?

3 Schönfinkelization

Let's take a universe where D_e contains three cats, Otto, Toot and Calvin. Otto loves Toot but not Calvin, and Toot loves Otto and Calvin, and Calvin doesn't love anybody, except, of course himself (all cats love themselves). In this universe, our characteristic function of the two-place predicate **love** looks like this, in table form:

14.

f_{love} =

<O,T> --> 1
<O,C> --> 0
<O,O> --> 1
<T,O> --> 1
<T,C> --> 1
<T,T> --> 1
<C,O>--> 0
<C,T>--> 0
<C,C>--> 1

If you Schönfinkel (or Curry) this two place function, what you do is turn it into a one-place function from entities to functions from entities to truth values. You can do it either way, taking the first argument of the pair as the single argument of the Schönfinkeled f_{love}, or taking the second argument as the argument of the Schönfinkeled f_{love}. The functions in 15 and 16 represent each of these strategies respectively, first left-to-right Schönfinkelization, and the second right-to-left Schönfinkelization.

14.

$$f_{love} = \left(\begin{array}{l} O \dashrightarrow \left(\begin{array}{l} T \dashrightarrow 1 \\ C \dashrightarrow 0 \\ O \dashrightarrow 1 \end{array}\right) \\ T \dashrightarrow \left(\begin{array}{l} O \dashrightarrow 1 \\ C \dashrightarrow 1 \\ T \dashrightarrow 1 \end{array}\right) \\ C \dashrightarrow \left(\begin{array}{l} O \dashrightarrow 0 \\ T \dashrightarrow 0 \\ C \dashrightarrow 1 \end{array}\right) \end{array}\right) \quad \text{Schönfinkeled left-to-right}$$

15.

$$f_{love} = \left(\begin{array}{l} O \dashrightarrow \left(\begin{array}{l} T \dashrightarrow 1 \\ C \dashrightarrow 0 \\ O \dashrightarrow 1 \end{array}\right) \\ T \dashrightarrow \left(\begin{array}{l} O \dashrightarrow 1 \\ C \dashrightarrow 0 \\ T \dashrightarrow 1 \end{array}\right) \\ C \dashrightarrow \left(\begin{array}{l} O \dashrightarrow 0 \\ T \dashrightarrow 1 \\ C \dashrightarrow 1 \end{array}\right) \end{array}\right) \quad \text{Schönfinkeled right-to-left}$$

Both ways, we end up with just 1-place functions, but of course, the value of the f_{love} function remains the same for each real-world situation, so you haven't changed its truth-functionality. You can do this for any n-place function. Which one corresponds to our function for **love** above?

Right, the right-to-left one, because the structure of the VP dictates that the V combine first with its object and then the result of that one is combined with the subject. Since the value of f_{love} = [[**love**]] when combined with an object x is yet another function, g_x, we can say that $f_{love}(x)=g_x$, and hence $g_x(y)= f_{love}(x)(y)$. (We could also notate this $[f(x)](y)$, to make it clearer that $f(x)$ is a function, but we don't really need to, because reading it that way is the only way that makes sense). $f_{love}(x)$= verb+object, and $f_{love}(x)(y)$ =

verb+object+subject. Here's where we have a function that looks sort of like the predicate logical formula Love(x,y), but in this case, the leftmost argument is the *object*, and the rightmost argument is the *subject.*

4 -notation

Switching from sets to characteristic functions was the big advance that let us make a compositional semantics out of first-order predicate logic. So far, we're writing functions in a fairly cumbersome way.

For [[**smoke**]], we've got the following notation:

16. [[**smoke**]] = f: D_e --> D_t
    for every x D_e, f(x) = 1 iff x smokes.

There's a shorthand way of writing this function:

17. f: [x: x D_e . x smokes]

This is the function that maps any x in D_e to 1 iff x smokes. In general, if we're using lambdas this way, the first element after the lambda-term is the domain, then there's the period, and then there's the condition that must obtain in order for the value of the function to be 1.

18. In general, if the lambda term denotes a truth value (i.e. if the function maps some type to truth values), then:
    [: .] is that function that maps any object such that to 1 iff .

Now, what about a function like this one:

19. f: |N --> |N
    for every x |N, f(x)=x+1

This is a function from natural numbers to natural numbers which gives the value, integer+1 for any x=integer. Now, here, we're not mapping onto truth values, we're mapping onto another type of thing. The lambda notation for this sort of function will look just the same as for the other sort of function:

20. f: [x: x |N . x+1]

This is a function that maps any x in the natural numbers to x+1. In this case, the item after the . in the lambda notation isn't giving a *condition* which must obtain if the value of the function is to be 1, rather, it's actually giving the *value of the function*.

21. In general, if the lambda term denotes a thing (any type of thing), then

[: .] is that function that maps every such that to .

So, we've got two ways of using the lambda-notation. In the first way, the item after the . gives the *truth condition* that must obtain for the function to give the value 1, and in the second way, the item after the . gives the actual value of the function. This may sound confusing, but it's really not. If is a sentence, then the function denoted by the lambda-term gives a truth value, and you read it as in 18; if is a thing (anything other than a sentence), then the function denoted by the lambda-term gives . (So, in 17, is "x smokes", which is a sentence, so is the condition that must apply for the function to give the value *true*, and in 20, is "x+1" which is a thing, so x+1 must be the value of the function).

Let's see if we can write our function for **love** in lambda-notation. We're going to have to use both conventions for reading lambda-terms to get it:

22. [[**loves**]] = f: D_e --> {g: D_e --> {0,1}}
for all x D_e , f(x) = g_x: D_e --> {0,1}
for all y D_e , g_x(y) = 1 iff y loves x

23. First, let's write the function g_x in lambda notation:

[[**loves**]] = f: D_e --> {g: D_e --> {0,1}}
for all x D_e , f(x) = [y: y D_e . y loves x]

The function g_x maps an entity to a truth value, so the lambda-term that denotes it is of the first type (in 18), and has a sentence for . Now, let's go ahead and write the function f in lambda notation:

24. [[**loves**]] = [x: x D_e . [y: y D_e . y loves x]]

Now, in this case,   is a thing, in fact, a member of the set of functions of type <e,t> (a member of $D_{<e,t>}$, so the value of this function is not a truth value, but that thing — to read this lambda-term, we use the convetion in 21.

Since lambda-terms are functions, they can take arguments:

25. [x: x D_e . x smokes](Ann)

is a sensible string, and has the value 1 iff Ann smokes.

What about

26. [x: x D_e . [y: y D_e . y loves x]](Toot)

This is a sensible string, and its value is 27.

27. [y: y D_e . y loves Toot]

Since 27 is itself a function, 28, is a sensible string, and has the value 1 iff Otto loves Toot:

28. [y: y D_e . y loves Toot](Otto)

Now, consider 29. Is it equivalent to 26?

29. [x: x D_e . [y: y D_e . y loves x](Toot)]

Nope, it's not. Here, Toot is the argument of the y term. What's the value of the y term with Toot as it's argument? It's "1 iff Toot loves x". That means that 29 can be written as:

30. [x: x D_e . Toot loves x]

(this will switch our lambda-term x to something that needs to be read by the first convention from 18) which says, this is a function which is true of an entity x iff Toot loves x.
Which of the following is equivalent to which?

31. (a) [x: x D_e . [y: y D_e . y loves x](Toot)](Otto)

(b) [x: x D_e . [y: y D_e . y loves x]](Toot)(Otto)

(c) [x: x D_e . [y: y D_e . y loves x]](Otto)(Toot)

(d) [x: x D_e . [y: y D_e . y loves x](Otto)](Toot)

32. *More shorthand*:
Sometimes the domain condition is just shovelled right in with the lambda-term:
[x D_e . x smokes]

This shouldn't cause any confusion, as long as you're careful with your brackets (cf. H&K p.38 ex (15)).

Homework:

So far, the semantic component H&K give consists of the following:

A: Possible semantic values, or *types*

i) elements of D, the set of individuals
ii) elements of {1,0}, the set of truth values
iii) functions from D --> {1,0}
iv) functions from { {1,0}x{1,0} } --> {1,0}
(*we made these up for the binary connectives in 5 & 6 above)*

B. Lexical entries

(i) [[**Ann**]] = Ann
(ii) [[**Joe**]] = Joe
...etc. for other proper names

(iii) [[**smokes**]] f: D --> {1,0}
For all x D, f(x) = 1 iff x smokes

(iv) [[**snores**]] f: D --> {1,0}
For all x D, f(x) = 1 iff x snores

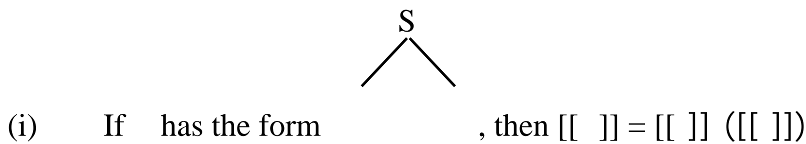
(v) [[**loves**]] = f: D_e --> {g: D_e --> {0,1}}
for all x D_e , f(x) = g_x: D_e --> {0,1}
for all y D_e , g_x(y) = 1 iff y loves x

(vi) [[**it-is-not-the-case-that**]] = f: {0,1} --> {0,1}
For all x D_t, f(x)=1 iff x=0

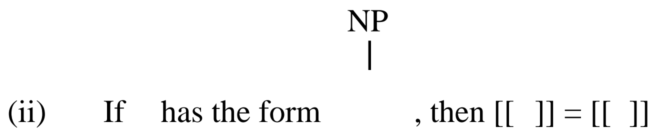
(vii) [[**and**]] = f: {0,1}x{0,1} --> {0,1}
For all <x,y> {0,1}x{0,1}, f(<x,y>)=1 iff x=1 and y=1

(viii) [[**or**]] = f: {0,1}x{0,1} --> {0,1}
For all <x,y> {0,1}x{0,1}, f(<x,y>)=1 iff x=1 or y=1

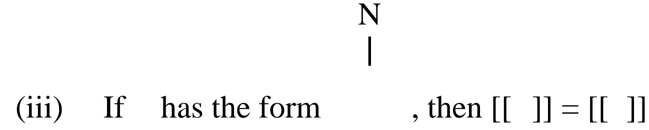
C. Rules for non-terminal nodes:



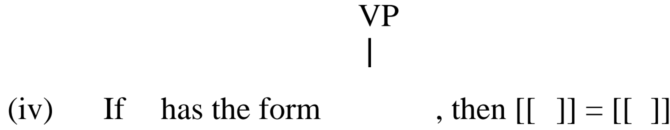
(i) If has the form , then [[ ]] = [[ ]] ([[ ]])



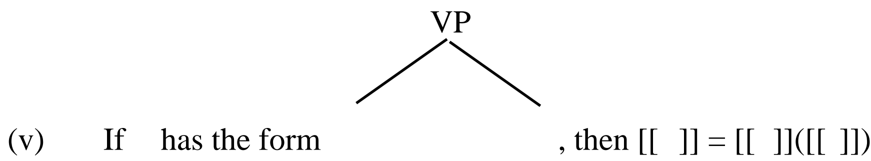
(ii) If has the form , then [[ ]] = [[ ]]



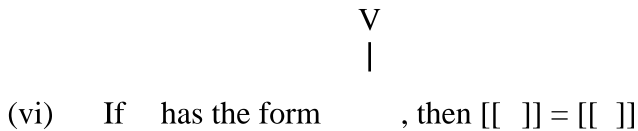
(iii) If has the form , then [[ ]] = [[ ]]



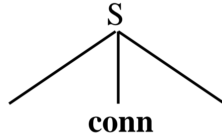
(iv) If has the form , then [[ ]] = [[ ]]



(v) If has the form , then [[ ]] = [[ ]]([[ ]])



(vi) If has the form , then [[ ]] = [[ ]]

(vii) If has the form 

, then [[ ]] = [[**conn**]](<[[ ]], [[ ]]>)

1. P. 31 Exercise 2.

2. P. 40 Exercise 3.

3. p. 32 Exercise 3 (this is *looong*. Do it thoroughly, though, in particular, be pedantic about (c). It'll pay off later, honest. Also, when you give the new lexical entry for **introduce**, give it in both -notation **and** the longer, easier-to-understand notation of the type in 22 above).

4. p. 39. Exercise 1.