Foundations of Semantics II: Composition

Rick Nouwen | R.W.F.Nouwen@uu.nl

February/March 2011, Gent

1 Compositionality

So far, we have only discussed what semantic meanings, truth-conditions, are, and what we are going to use to write them down. The goal of a formal semantic theory is to provide a mechanism that systematically assigns truth-conditions (or representations thereof) to natural language sentences. In contemporary semantics, we do this by means of a guiding principle (often attributed to Gottlob Frege):

> **The Compositionality Principle —** *The meaning of the whole is determined (solely) by the meaning of the parts and their mode of composition.*

The intuition behind this principle is that meanings are derived by means of a fully productive mechanism. That is, we do not learn the meanings of sentences, we *derive* them from what we know about the meanings of words. So, we understand what the example in (1) means because we are acquainted with the words in that sentence, even though this is the first time we have ever seen this particular sentence.

(1) The old cat sat in the middle of a huge square opposite a small church with a blue tower.

Discussion: one version of the compositionality principle states that the meaning of the whole is *exhaustively* determined by the meaning of the parts and their mode of composition. (The version with 'solely' in the presentation above). Can you think of examples where this particular principle appears to be too strong?

(If you have heard of *indexicals* before, think of examples with one or more indexicals in them.)

In semantics, we use compositionality as a general methodological principle. That is, we want our semantics to be compositional, or at least as compositional as possible. This means that given a surface structure and a lexicon, the semantics allows us to determine the truth-conditions of the structure. So, the semantics does not tell us what the truth-conditions of *John hates Mary* are in any direct way, but it will tell us how to derive these truth-conditions given the structure '[John [hates Mary]]' and the meaning of *John*, *hate* and *Mary*.

In terms of a referential theory of meaning, this means that the lexicon is responsible for stating the reference of individual words, while the compositional system is responsible for *deriving* the reference of complex expressions.

In the field, there are two styles of compositional semantics. The dominant style is to make use of a syntactic level of representation that functions as the input to semantic derivation called *logical form* (LF). In a way, this style has it that compositionality is *indirect*. That is, interpretation is not compositional with respect to surface structure, but rather with respect to an intermediate level. According to proponents of *direct compositionality*, such an intermediate level is unnecessary. Directly compositional semantics strives to show that the level of logical form can be eliminated. (Cf. figure 1.)

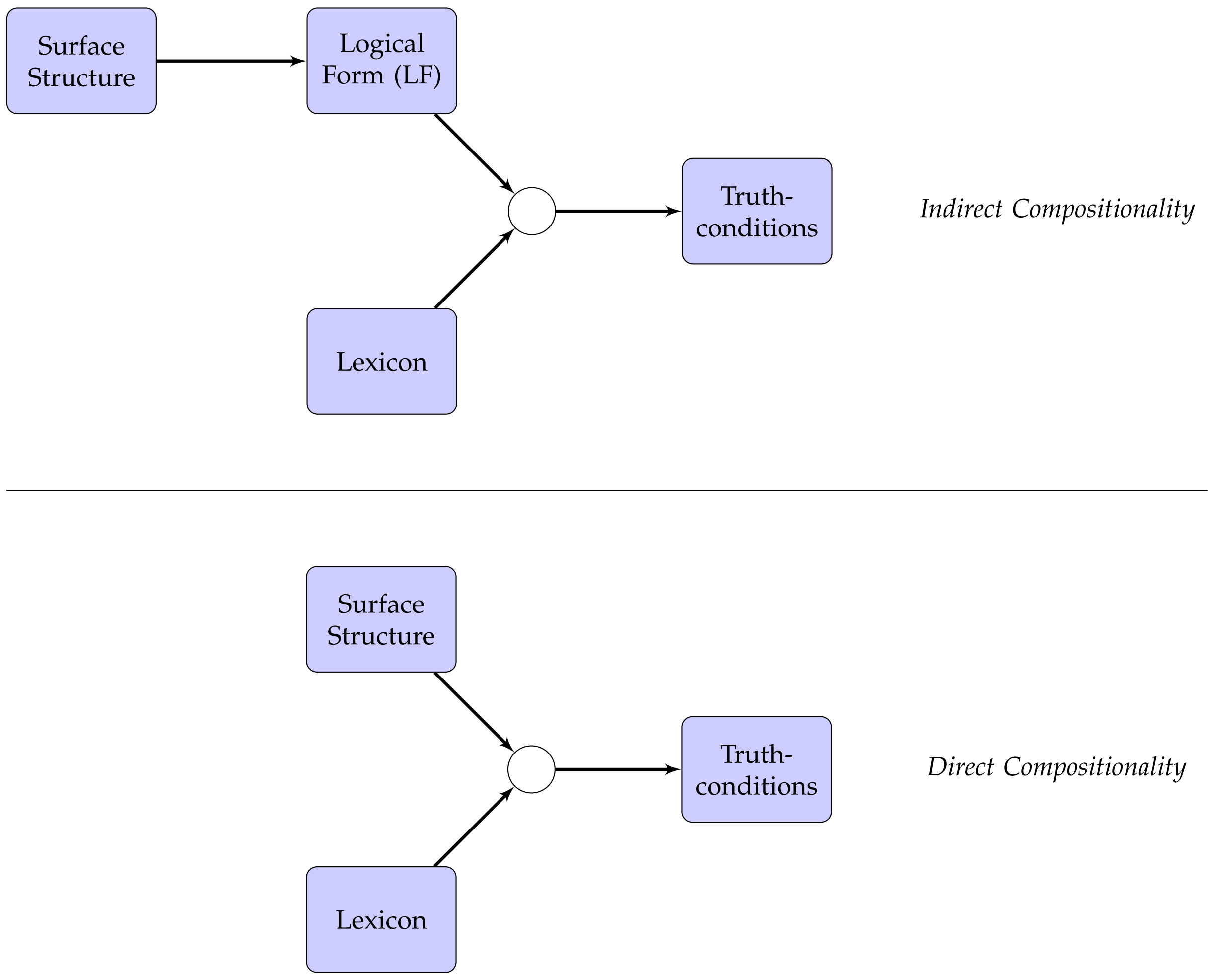


Figure 1: Direct versus indirect compositionality

In what follows, we shall focus on the indirectly compositional approach to natural language semantics. The main reason for this is that it is often insightful to use a level of logical form, especially since it allows for easy scrutiny of the relation between semantic and syntactic mechanisms. Let me illustrate with a look ahead. Consider the following example:

(2) Some woman loves every man.

This example is ambiguous. Here are two predicate logical formulae corresponding to the two readings:

(3) a. $\exists x[woman(x) \wedge \forall y[man(y) \rightarrow love(x, y)]]$
(there is some woman such that she loves every man; that is, she is indiscriminate in love)
b. $\forall y[man(y) \rightarrow \exists x[woman(x) \wedge love(x, y)]]$
(for every man in the world, we can find at least one woman who loves this man)

It is the reading in (3-b) that is a challenge to account for. As we will see in detail later, the reason is because this reading seems to involve an ordering of the quantifiers in this sentence that does not correspond to the surface ordering. In (3-b), the existential quantifier is in the scope of the universal quantifier.

It is common in indirect compositionality approaches to account for (3-b) by making use of a form of movement. So, the two paraphrases in (3) correspond to two different logical forms, one in which no movement took place and one in which the object quantifier moves to a higher position.

(4)
  a. [some woman [loves [every man]]]
  b. [every man [some woman [loves __]]]

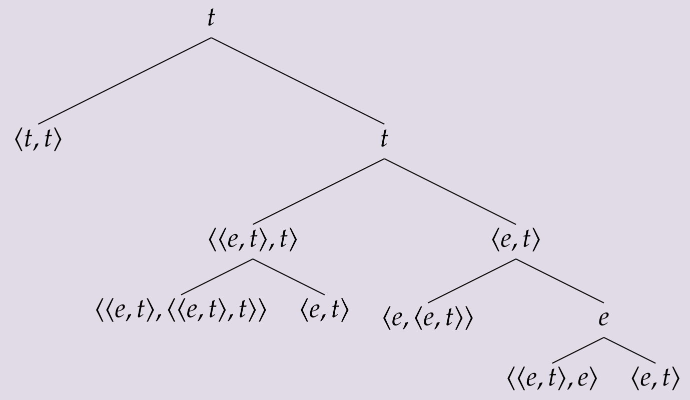
On the basis of these two logical forms, we can now compositionally derive two (distinct) truth-conditions, namely (3-a) for (4-a) and (3-b) for (4-b). (We'll see how this works in detail later in the course). Crucial, however, is that prior to this compositional derivation, we needed to perform a syntactico-semantic operation on the purely representational level of LF.

In contrast, a directly compositional approach would account for the same ambiguity without having to resort to movement, or any other method that necessitates a level like logical form. The easiest way to do this (but there are important and more intricate techniques I cannot go into now) is by saying that transitive verbs like *love* are ambiguous with respect to which argument has scope over which argument.

2 Functions and Types

Puzzle— There is a logic behind the labeling of the nodes in this tree. Can you figure it out?

(5)


t
$\langle t,t\rangle$
t
$\langle\langle e,t\rangle,t\rangle$
$\langle e,t\rangle$
$\langle\langle e,t\rangle,\langle\langle e,t\rangle,t\rangle\rangle$
$\langle e,t\rangle$
$\langle e,\langle e,t\rangle\rangle$
e
$\langle\langle e,t\rangle,e\rangle$
$\langle e,t\rangle$

The mechanism illustrated in the tree in (5) is that of *function application*, the main mode of composition for meaning derivation. Before we can explain the details of this mechanism and what (5) is meant to represent, we have to make sure we understand what a *function* is.

2.1. Functions and Types

An example of a function is f in (6), which you may remember from a distant or not so distant school past:

(6) $f(x) = x + 1$

The function f adds 1 to any *argument* you feed the function. So, $f(3)$ returns $3 + 1$ (i.e. 4), $f(352)$ returns 353, etc. This is an instance of function application: you combine the function with an argument and get in return an output value. This value is derived from the argument in a systematic way, in this case by adding 1 to the argument. One way to represent the relation between function, argument and function value is by means of a tree structure:

(7)
$$\begin{tikzcd}
& 353 \arrow[dl, dash] \arrow[dr, dash] & \\
f & & 352
\end{tikzcd}
\qquad
\begin{tikzcd}
& \text{function value} \arrow[dl, dash] \arrow[dr, dash] & \\
\text{function} & & \text{argument}
\end{tikzcd}$$

Here's another example from arithmetic:

(8)
$$\begin{tikzcd}
& 4 \arrow[dl, dash] \arrow[dr, dash] & \\
\surd & & 16
\end{tikzcd}$$

As in predicate logic, it makes sense to distinguish variables from constants in the definition of functions. In the definition of f above, there is mention of 'x' and of '1'. Clearly, x is a variable. It is a place holder for whatever argument we are going to combine the function with; it does not have a fixed reference. In contrast, 1 is a constant: it refers to an entity with a unique set of properties.

The function f above targets arithmetic entities such as integers or reals. Since the operation + only makes sense with respect to numbers, it would be inappropriate to apply f to an individual like John. So, we know that the arguments f takes are of a particular *type*, namely that of numbers. Also, we know that any function value that results from applying f to an argument is also of that type. But now consider function g:

(9) $g(x) = x$'s age

Numbers don't have an age: arguments of g must be living things. The function value resulting from g being applied to something alive is a number, and so we conclude that g takes arguments of type *living things* to return function values of type *numbers*. On the basis of this, we can specify the *type of the function*. We do this by combining the type of the arguments and the type of the function values in an ordered pair. For instance, the type of g is ⟨living things,numbers⟩. The type of f is ⟨numbers,numbers⟩.

We indicate types with colons, as follows:

(10) a. f : ⟨numbers,numbers⟩
b. 324 : numbers
c. John : living things
d. g : ⟨living things,numbers⟩

The general workings of functions and types is now summarised as follows. If I have a function φ of type $\langle\alpha,\beta\rangle$, then function applying φ to A of type α will result in a function value of type β. Schematically,

(11)
$$\begin{tikzcd}
& \varphi(A){:}\beta \arrow[dl, dash] \arrow[dr, dash] & \\
\varphi : \langle\alpha,\beta\rangle & & A : \alpha
\end{tikzcd}$$

As we will see, the schema in (11) is the main driving force behind compositional interpretation. However, for the purpose of linguistic semantics, we will have use no use for types like *living things* or *numbers*. In fact, much of formal semantics revolves around two types: (i) entities, notated as *e*, and (ii) truth-values, notated as *t*. There are only two objects of type *t*, namely *true* and *false*, often also notated as *1* and *0*, respectively. There are many (probably infinitely many) objects of type *e*, and they include living things, dead things, inanimate objects, thoughts, numbers, experiences, etc., roughly anything for which there exists a countable noun.

(12) Examples of the two main types used in formal semantics

	e	*t*
instantiations:	any entity	just 1 and 0
corresponding natural language expressions:	proper names definite descriptions examples: *John, Aristotle, the key on my desk, the pope, the brilliant idea I just had, my next-door neighbour*	clauses

In language, referential DPs are an example of expressions (with a meaning) of type *e*. Declarative sentences are an example of expressions (with a meaning) of type *t*. However, in formal semantics, we can give any word or constituent a type. So what is the type of a noun? Or of a verb? The answer is that such expressions have a functional type.

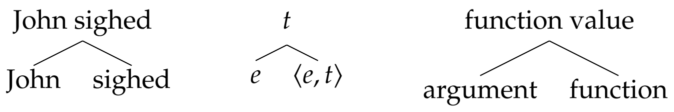
Consider the simple example in (13).

(13) John sighed.

What does *sighed* contribute to the truth-conditions of this sentence? Well, a sentence with *sighed* as its main verb is true if the subject sighed and false otherwise. (Caveat: this and what follows is an enormous oversimplification, since we are ignoring tense-marking and assuming that this marking is part of lexical semantics.)

(14) The meaning of *sighed*
  a. If the subject sighed, return *true*
  b. If the subject didn't sigh, return *false*

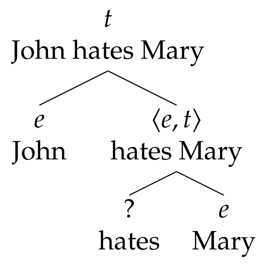
In other words, the meaning of *sighed* is a function. It takes arguments of type *e* and returns a value of type *t*. *Sighed* expresses a function of type $\langle e, t\rangle$. On the basis of this view on verb meaning, we can present our first compositional interpretation.

(15)


Let us now look at the type of a verb phrase. Consider the VP of (16), *hates Mary*.

(16) John hates Mary.

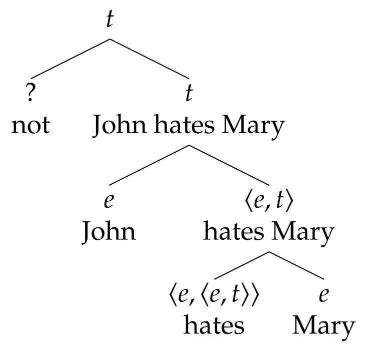
As with *sigh*, we can assume that *hates Mary* expresses a $\langle e,t\rangle$-function: it maps subjects who hate Mary to true and other subjects to false. But what is the type of the verb *hate* itself? First of all, notice that we can present a partial depiction of the types involved in (16).

(17)


From (17) and the general composition schema in (11),[1] we can deduce the type of *hates*. This type is quite complex: it takes an entity (the object) to return something which itself is a function (the verb phrase). To be precise, *hate* expresses a function of type $\langle e,\langle e,t\rangle\rangle$. This type makes sense, too:

(18) The meanig of *hates*
is the function that applies to an entity (the object) to return a new function which maps a second entity (the subject) to true if the subject hates the object and to false otherwise

Let's look at another example, negation. Assume that a sentence like *John doesn't hate Mary* has the structure in (19). That is, for expository purposes we ignore the auxiliary and we assume we understand how and why at logical form negation ends up high in the structure.

(19)


As before, the composition schema in (11) dictates the type of *not*: $\langle t,t\rangle$. This, too, is intuitive, for negation takes a sentence and returns its denial. If the sentence it applies to is true, then the value it returns is false. Applying negation to a false sentence, yields true.

Negation makes it possible to discuss a potential terminological confusion. According to the $\langle t,t\rangle$ type, the arguments of negation are of type t. In semantics, and in what follows, it is important not to confuse arguments of functions with the arguments of syntactic theory, i.e. the things that get theta-marked. When the transitive verb *hates* combines with *Mary* to form a verb phrase *hates Mary*, then *Mary* is an argument in two senses. Syntactically, hate assigns a theta role to *Mary*, which makes Mary an argument. Moreover, in a compositional derivation, the meaning of *Mary* is an argument to the function expressed

[1]Note that we have to assume that it is irrelevant whether the function is to the left or the right of its argument. So, other than (11) might suggest, the order of function and argument is not important.

by the verb. In (19), the meaning of the sentence *John hates Mary* is an argument of the function expressed by negation, but it is not so obviously an argument in the other sense of the word. It is important to be aware of these two notions, especially since, as we will see later, it turns out that verbs and verb phrases are often arguments in a compositional derivation.

2.2. The $\langle e, t\rangle$ type

The type $\langle e, t\rangle$ is the type of meanings that are functions mapping entities to truth-values. Such functions are domain-splitters: they assign true to one part of the domain and false to the rest. This is exactly the behaviour of sets, for they can be thought of as mapping their elements to true and the rest to false.

Given the close correspondence between predicates and sets, it should come as no surprise that the type $\langle e, t\rangle$ is related to the unary predicates of predicate logic. This is because these predicates call for an argument to return a proposition (i.e. something with a truth value). For instance, in a proposition *asleep*(j), the predicate *asleep* was combined with its argument j.

In natural language, many kinds of expressions correspond to the $\langle e, t\rangle$ type. (This should not be a surprise, since we've already remarked that the predicates of predicate logic correspond to a multitude of linguistic entities.) Among the grammatical entities that are usually expected to have meanings of type $\langle e, t\rangle$ are: nouns, adjectives, intransitive verbs, verb phrases, relative clauses.

Here are some examples of how the $\langle e, t\rangle$ types, sets and properties all amount to the same thing:

(20) *red*
  a. refers to a meaning of type $\langle e, t\rangle$
  b. refers to a property, namely the property of being red
  c. refers to a set, namely the set of red entities
  d. corresponds to a unary predicate constant in predicate logic: *red*

(21) *student*
  a. refers to a meaning of type $\langle e, t\rangle$
  b. refers to a property, namely the property of being a student
  c. refers to a set, namely the set of students
  d. corresponds to a unary predicate constant in predicate logic: *student*

(22) *hate Mary*
  a. refers to a meaning of type $\langle e, t\rangle$
  b. refers to a property, namely the property of hating Mary
  c. refers to a set, namely the set of individuals who hate Mary

(23) *who killed John*
  a. refers to a meaning of type $\langle e, t\rangle$
  b. refers to a property, namely the property of having killed John
  c. refers to a set, namely the set of individuals who killed John

Exercise— What is the type of *the* in *the man snored*?

3 The λ-calculus

Note that I said that an adjective like *red* or a noun like *student* 'corresponds' to a predicate, not that it referred to one. This is because what such expressions refer to are $\langle e, t\rangle$-functions and a predicate constant is not such a function, but is rather a name for something that in predicate logic is interpreted as such. There is a way to extend predicate logic, however, to express directly the relation between types and meanings. This is done by means of the so-called *lambda calculus*.

To get the intuition, let us first consider a simple unary predicate like *red*. In predicate logic, *red* is not by itself a very meaningful expression. It does not specify, for instance, how many arguments it needs in order to form a proposition. With the lambda calculus we can express precisely what is needed to saturate the arguments of predicates. More generally, with the lambda calculus one has a language that can express functions and their application, in a very precise way.

Let us start with the arithmetic example that we used to introduce the notion of functions.

(24) $f(x) = x + 1$

The whole statement in (24) defines a function and gives it a name, f. There is a way to define the same function, however, without having to give it a name. This we do with a lambda operator. The expression in (25)– something like this is called a *lambda term* – is itself the function that we called f in (24).

(25) $\lambda x.x + 1$

The way to read it is as follows. The λx before the full stop indicates that this function is awaiting an argument. The right-hand side of the term specifies what the function outputs when it gets this argument, namely that it adds 1 to it.

Note that the variable name is immaterial, it is simply there to keep track of the hypothetical argument. So, (26) specifies exactly the same function as (24) and (25).

(26) $\lambda y.y + 1$

(Note that $\lambda y.x + 1$ specifies a different function, for it awaits an argument, which we are going to call y and then adds 1 to x, whatever that may be.)

It is possible to have multiple lambdas, expressing multiple arguments. For instance, (27) is a function that asks for two arguments and returns the value corresponding to the sum of the two arguments:

(27) $\lambda x.\lambda y.x + y$

The way to read such functions with multiple arguments is as a complex function. That is, (27) asks for an argument, say 3, and returns the function $\lambda y.3 + y$. This new function can then take a novel argument, say 5, to yield 3 + 5.

This shows you how lambda terms are used. They are functions. When applied to an argument, the variable that labels the lambda is substituted by this argument. This latter step is called β-reduction.

(28) $\lambda y.y + 1\ (4)$ function application
$= 4{+}1$ beta reduction

In semantics, we use the lambda calculus to express functions that involve predicate logical propositions. For instance, the adjective *red* refers to the following function:

(29) $\lambda x.red(x)$

This function awaits an argument (x) and returns a predicate logical proposition that consists of the predicate *red* with the function's argument as its argument. So, if we apply the function in (29) to j, through beta reduction we will end up with the proposition $red(j)$. If we apply the function in (29) to y, we end up with $red(y)$, etc.

The benefit of the lambda calculus is that we now have a way to express complex properties / sets. Say, for instance, that our predicate logic has among its unary predicate constants the predicates *lazy* and $professor$, then (30) expresses the property of being a lazy professor (i.e. the set of lazy professors):

(30) $\lambda x.lazy(x) \wedge professor(x)$

If we apply (30) to an argument it will return a proposition that is only true if the argument is both lazy and a professor. So apply it to *John*, j, a lazy student, and the output proposition will be $lazy(j) \wedge professor(j)$, which will be false; apply it to *Mary*, m, a lazy professor, and the output is $lazy(m) \wedge professor(m)$, which will be true.

Here is an example of a complex verb phrase meaning:

(31) $\lambda y.hate(y, j) \wedge \neg hate(y, m)$

This function takes an argument (y) and returns a proposition that says that this argument hates j, but not m. If j is called John and m Mary, then this function could correspond to the verb phrase *hates John but not Mary*.

Exercise— Paraphrase the following functions

(32) a. $\lambda y.hate(y, j) \wedge despise(y, j)$
b. $\lambda y.\forall x[student(x) \rightarrow hate(y, x)]$
c. $\lambda y.\forall x[student(x) \rightarrow hate(x, y)]$

4 Lambdas and Types

It is quite easy to recognise the type of function that takes the form of a lambda-term.

(33) The type of a lambda term $\lambda x.\varphi$ is $\langle$the type of x , the type of $\varphi\rangle$.

Although this is not set in stone, it is good practice to reserve lower case letters for e-type constants and variables (where the last bit of the alphabet is usually reserved for variables and the rest for constants). An exception is usually the letter p, which is often used for things of type t. Upper case letters usually stand for things of a higher order, like $\langle e, t\rangle$-type functions etc. We assume that predicate logical propositions are of type t. In general, the type of something will usually become clear from the context. For instance, if a lambda term contains the proposition $hate(j, X) \wedge lazy(X)$, then it should be clear that, despite its upper case, X is meant to be of type e, for otherwise it would not be able to be a term argument to a predicate.

Some examples:

(34) $\underbrace{\lambda x}_{e} . \underbrace{red(x)}_{t}$ is of type $\langle e,t\rangle$

$\underbrace{\lambda x}_{e} . \underbrace{\neg red(x) \wedge marble(x)}_{t}$ is of type $\langle e,t\rangle$

$\underbrace{\lambda x}_{e} . \underbrace{x}_{e}$ is of type $\langle e,e\rangle$

$\underbrace{\lambda p}_{t} . \underbrace{\neg p}_{t}$ is of type $\langle t,t\rangle$

$\underbrace{\lambda x}_{e} . \underbrace{\underbrace{\lambda y}_{e} . \underbrace{hate(y,x) \wedge despise(y,x)}_{t}}_{\langle e,t\rangle}$ is of type $\langle e,\langle e,t\rangle\rangle$

5 Putting it all together

In this section we will provide a very detailed analysis of a single simple example to on the one hand review all the formal concepts and tools we introduced, while on the other explain how the compositional derivation of meaning works.

(35) John hates Mary.

The semantic meaning of this sentence is a set of truth-conditions. The preferred way to express these is by means of predicate logic. For instance, given a predicate logical language that includes the individual constants j and m and the binary predicate *hate*, which are interpreted in such a way that j is John, m is Mary and *hate* the relation that specifies who hates who in each possible world, then the truth-conditions for (35) are as follows.

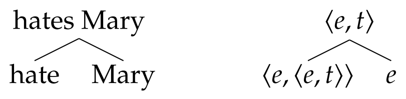
(36) $hate(j,m)$

These truth-conditions need to be derived compositionally, however. We do this as follows. In the lexicon we store the reference of words. We do this by assigning words either lambda terms or e-type expressions. We borrow the $[\![\ ]\!]$ notation that we introduced in predicate logic to express the interpretation relation.

(37) a. $[\![\text{John}]\!] = j$
b. $[\![\text{Mary}]\!] = m$
c. $[\![\text{hates}]\!] = \lambda x.\lambda y.hate(y,x)$

Note that we do not say that the word *hates* refers to the predicate constant *hate*. The reason for this is that *hate* is not explicit about its arguments, while $\lambda x.\lambda y.hate(y,x)$ is. In fact, the lambda term dictates the order in which the arguments are received: first the theme argument, then the experiencer argument. This makes sure that the object becomes the theme and the subject the experiencer.

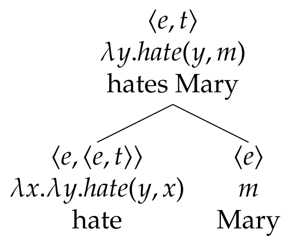
On the basis of these references of the individual parts of the sentence, we can built the references of the complexes. Let us start with the verb phrase. We have already seen what to expect of the type of the verb phrase:

(38)


The meaning of the composition of two expressions results from function applying the meaning of one of these expressions to the meaning of the other. In *hates Mary* we apply the $\langle e, \langle e, t\rangle\rangle$-function referred to by the verb to the e-type argument. The result is a new function of type $\langle e, t\rangle$. These types reflect the following composition of meanings:

(39) $\lambda x.\lambda y.hate(y, x)\ (m)$ function application
$= \lambda y.hate(y, m)$ beta reduction

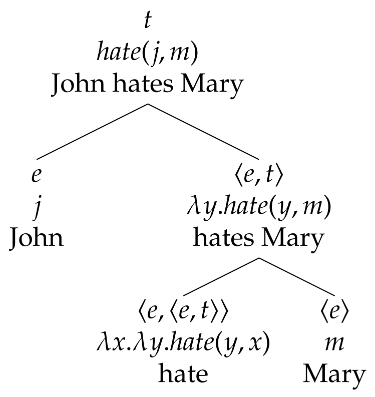
We can put all things together in a tree representation:

(40)


The verb phrase is a function of type $\langle e, t\rangle$ and, so, if it takes the reference of the subject as its argument, the sentence will refer to something of type t. The function application involved is straightforward:

(41) $\lambda y.hate(y, m)\ (j)$ function application
$= hate(j, m)$ beta reduction.

The whole derivation is now summed up as follows:

(42)


Note that we can now interpret any node in the tree. The leaves are interpreted via the lexicon, the rest via function application. For instance:

(43) $[\![[_{\mathrm{VP}} \text{ hates Mary}]]\!] = \lambda x.\lambda y.hate(y, x)(m) = \lambda y.hate(y, m)$

6 Interlude: the lambda calculus and set theory

Heim and Kratzer (1998) do it differently, they avoid predicate logic and instead talk about functions and set theory directly.

(44) $$[\![\text{smokes}]\!] = \begin{bmatrix} \text{Ann} & \rightarrow & 1 \\ \text{John} & \rightarrow & 1 \\ \text{Mary} & \rightarrow & 0 \end{bmatrix}$$

(45) a. $[\![\text{John}]\!] = \textit{John}$
b. $[\![\text{Mary}]\!] = \textit{Mary}$

(46) $$[\![\text{hates}]\!] = \begin{bmatrix} \textit{Ann} & \rightarrow & \begin{bmatrix} \textit{Ann} & \rightarrow & 0 \\ \textit{John} & \rightarrow & 1 \\ \textit{Mary} & \rightarrow & 0 \end{bmatrix} \\ \\ \textit{John} & \rightarrow & \begin{bmatrix} \textit{Ann} & \rightarrow & 0 \\ \textit{John} & \rightarrow & 1 \\ \textit{Mary} & \rightarrow & 1 \end{bmatrix} \\ \\ \textit{Mary} & \rightarrow & \begin{bmatrix} \textit{Ann} & \rightarrow & 0 \\ \textit{John} & \rightarrow & 0 \\ \textit{Mary} & \rightarrow & 0 \end{bmatrix} \end{bmatrix}$$

(47) $[\![\text{John smokes}]\!] = [\![\text{smokes}]\!]([\![\text{John}]\!]) = 1$

(48) $$[\![\text{hates Mary}]\!] = [\![\text{hates}]\!]([\![\text{Mary}]\!]) = \begin{bmatrix} \textit{Ann} & \rightarrow & 0 \\ \textit{John} & \rightarrow & 0 \\ \textit{Mary} & \rightarrow & 0 \end{bmatrix}$$

(49) $$[\![\text{John hates Mary}]\!] = [\![\text{hates Mary}]\!]([\![\text{John}]\!]) = \begin{bmatrix} \textit{Ann} & \rightarrow & 0 \\ \textit{John} & \rightarrow & 0 \\ \textit{Mary} & \rightarrow & 0 \end{bmatrix}(\textit{John}) = 0$$

In our approach we would use predicate logic. Say, we have predicate *smoke* (unary) and *hate* (binary) and the individual constants *j*, *m* and *a*. The following interpretation of these constants is equivalent to the above.

(50) $[\![\textit{smoke}]\!] = \{\textit{Ann}, \textit{John}\}$
$[\![\textit{hate}]\!] = \{\langle \textit{John}, \textit{Ann}\rangle, \langle \textit{John}, \textit{John}\rangle, \langle \textit{Mary}, \textit{John}\rangle\}$
$[\![j]\!] = \text{John}$
$[\![m]\!] = \text{Mary}$
$[\![a]\!] = \text{Ann}$

The lexicon would now contain the following interpretations:

(51) $[\![\text{smokes}]\!] = \lambda x.\textit{smoke}(x)$
$[\![\text{hates}]\!] = \lambda x.\lambda y.\textit{hate}(y, x)$
$[\![\text{John}]\!] = j$
$[\![\text{Mary}]\!] = m$
$[\![\text{Ann}]\!] = a$

Note the different levels:

(52)

natural language	predicate logic	the model
smokes	$smoke$	$\{Ann, John\}$
hates	$hates$	$\{\langle John, Ann\rangle, \langle John, John\rangle, \langle Mary, John\rangle\}$
John	j	$John$
Mary	m	$Mary$
Ann	a	Ann
Ann smokes	$smoke(a)$	1
Ann hates John	$hate(a, j)$	0

7 Definite descriptions

So far, we have only been able to interpret one kind of natural language expression of type e, namely proper names. Definite descriptions are also of this basic type. This is evident from the fact that definite descriptions can have the same reference as a proper name.

(53) a. Joseph Ratzinger
b. the pope

(54) a. Barack Obama
b. the president of the US

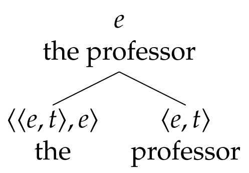
(55) a. Mary
b. the woman

(56) a. John
b. the best student in my class

There are important semantic differences between proper names and definite descriptions that we will avoid for now. These involve observations like the following: Joseph Ratzinger will always be Joseph Ratzinger, but the pope will not always be Joseph Ratzinger. We will discuss such matters further when we turn to *intensionality*. For now, we simply focus on the observation that proper names and definite descriptions both refer to entities of type e.

The definite article combines with nouns to create a definite description of type e. We have claimed above that nouns are of type $\langle e, t\rangle$. That is, *professor* expresses the property of being a professor / the set of professors. This means that in a DP like *the professor*, the definite article takes the $\langle e, t\rangle$-reference of the noun as an argument, to return something of type e. In other words, *the* must have a meaning of type $\langle\langle e, t\rangle, e\rangle$. Schematically,

(57)



We have deduced the type of the definite article, but what about its actual meaning? A definite description like *the professor* refers to the (contextually) unique entity that has the property of being a professor. In predicate logic, there exists a notation for such descriptions that we have not yet introduced. So far, only variables and constants could be terms and thereby arguments of predicates. There is a 3rd kind of term however, so-called ι-terms. The expression in (58) is a way of referring, in predicate logic, to the unique entity with the property of being a professor.

(58) $\iota x.professor(x)$

Note that the expression in (58) is a term, so as a whole it can be the argument of a predicate. For instance, (59) says that the professor is lazy but not tired.

(59) $lazy(\iota x.professor(x)) \wedge \neg tired(\iota x.professor(x))$

There can be quite a complexity to ι-terms. For instance, (60) refers to the professor who admires every student.

(60) $\iota x.professor(x) \wedge \forall y[student(y) \rightarrow admire(x,y)]$

For clarity, we will sometimes indicate the scope of the ι-operator with some extra parenthesis, as in:

(61) $\iota x.[professor(x) \wedge \forall y[student(y) \rightarrow admire(x,y)]]$

Exercise— Paraphrase the following predicate-logical formulae

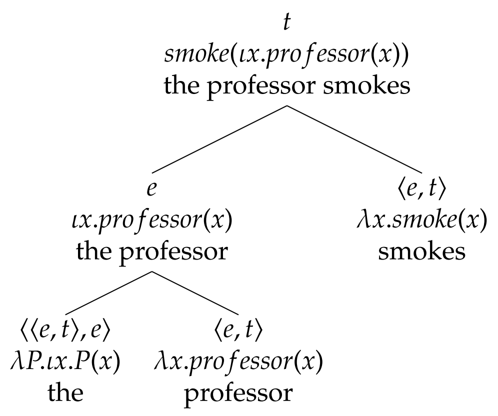
(62) a. $\forall z[student(z) \rightarrow admire(z, \iota x.professor(x) \wedge \forall y[student(y) \rightarrow admire(x,y)])]$
b. $\neg professor(\iota x.professor(x))$
c. $admire(m, \iota x.boy(x) \wedge \neg admire(x,m))$

Using the ι-operator, the definite article is interpret as follows:

(63) $[\![\text{the}]\!] = \lambda P.\iota x.P(x)$

This results in:

(64)



⋆

We end this handout with two analyses from the Heim and Kratzer text book that have become especially popular, and which introduce some particularities of the composition process. (In what follows, we stick with our approach involving predicate logic, however.)

See Heim and Kratzer 1998 chapters 1–5.

⋆

8 Adjectives

As we have said before, the meaning of expressions of many categories are of the $\langle e,t\rangle$-type. Among them are adjectives. The idea is that an adjective like *lazy* refers to the property of being lazy, i.e. the set of lazy individuals:

(65) $[\![\text{lazy}]\!] = \lambda x.lazy(x)$

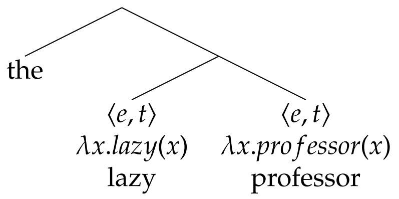
Consider an example like (66).

(66) the lazy professor

Ideally, we would want to interpret the NP *lazy professor* as in (67), for combined with the semantics of the definite article introduced in the previous section, this would result in $\iota x.lazy(x) \wedge professor(x)$ (the unique lazy professor).

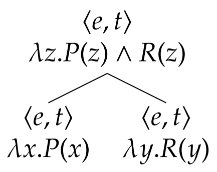
(67) $\lambda x.lazy(x) \wedge professor(x)$

A problem occurs, however. Consider the tree in (68).

(68)


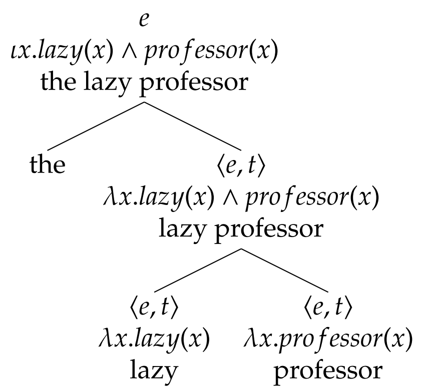
This tree presents a *type clash*. There are two sister nodes that are both of type $\langle e,t\rangle$. The result is that no function application can take place.

The solution that Heim and Kratzer 1998 offer is to introduce a novel composition rule. That is, apart from function application, they introduce a mode of composition called *predicate modification*. Schematically, it looks like (69).

(69)


predicate modification

Applied to *the lazy professor* this yields:

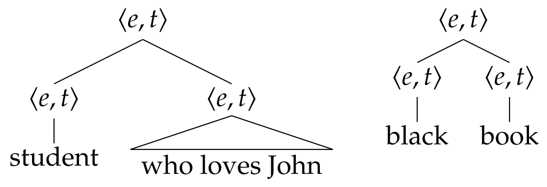
(70)


Exercise— Predicate modification use of the fact that it appears that attributive adjectives are interpreted *intersectively*. That is, *John is a lazy professor* entails both that *John is lazy* and that *John is a professor*. The assumption that all adjectives are interpreted in this way is wrong, however. The following examples illustrate that. Discuss.

(71) a. former president
b. alleged criminal
c. stone lion
d. good cook

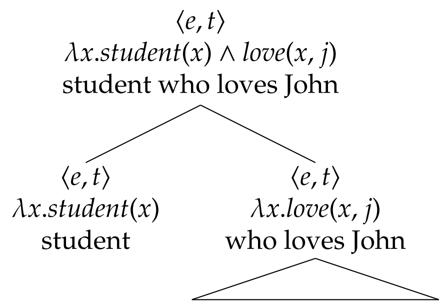
9 Relative Clauses

Many semanticists assume that relative clauses express predicates of type $\langle e,t\rangle$. That is, *who loves John* expresses the set of individuals who love John. This makes the case of relative clauses analogous to that of adjectives, who have this type $\langle e,t\rangle$ too. Both relative clauses and adjectives combine with nouns and both poses a type-logical problem discussed in the previous section.

(72)


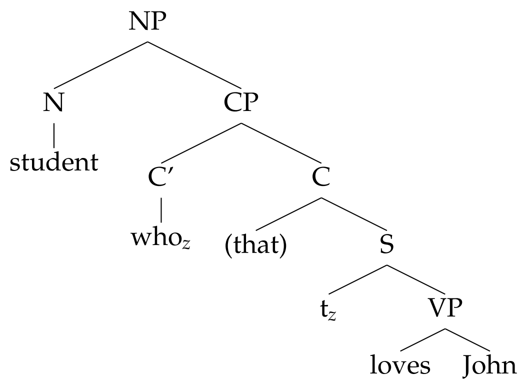
Given predicate modification, and given our assumption that *who loves John* refers to something of type $\langle e, t\rangle$, our analysis of *student who loves John* will be as follows.

(73)



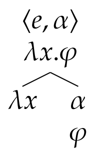
But how do we get to the $\langle e, t\rangle$-type reference of the relative clause? Here we follow (Heim and Kratzer 1998) again, in adopting the following LF structure.

(74)



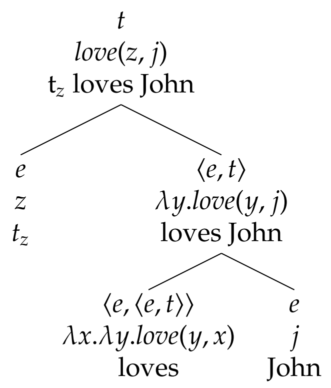
Here we introduce a couple of new aspects to LF interpretation: *indices* and *traces*. The interpretation of (74) makes use of the following assumptions:

— traces correspond to variables of type *e*
— relative pronouns introduce a λ abstraction
— co-indexed nodes share the same choice of variable
— nodes that introduce λ abstraction compose schematically as in the tree:



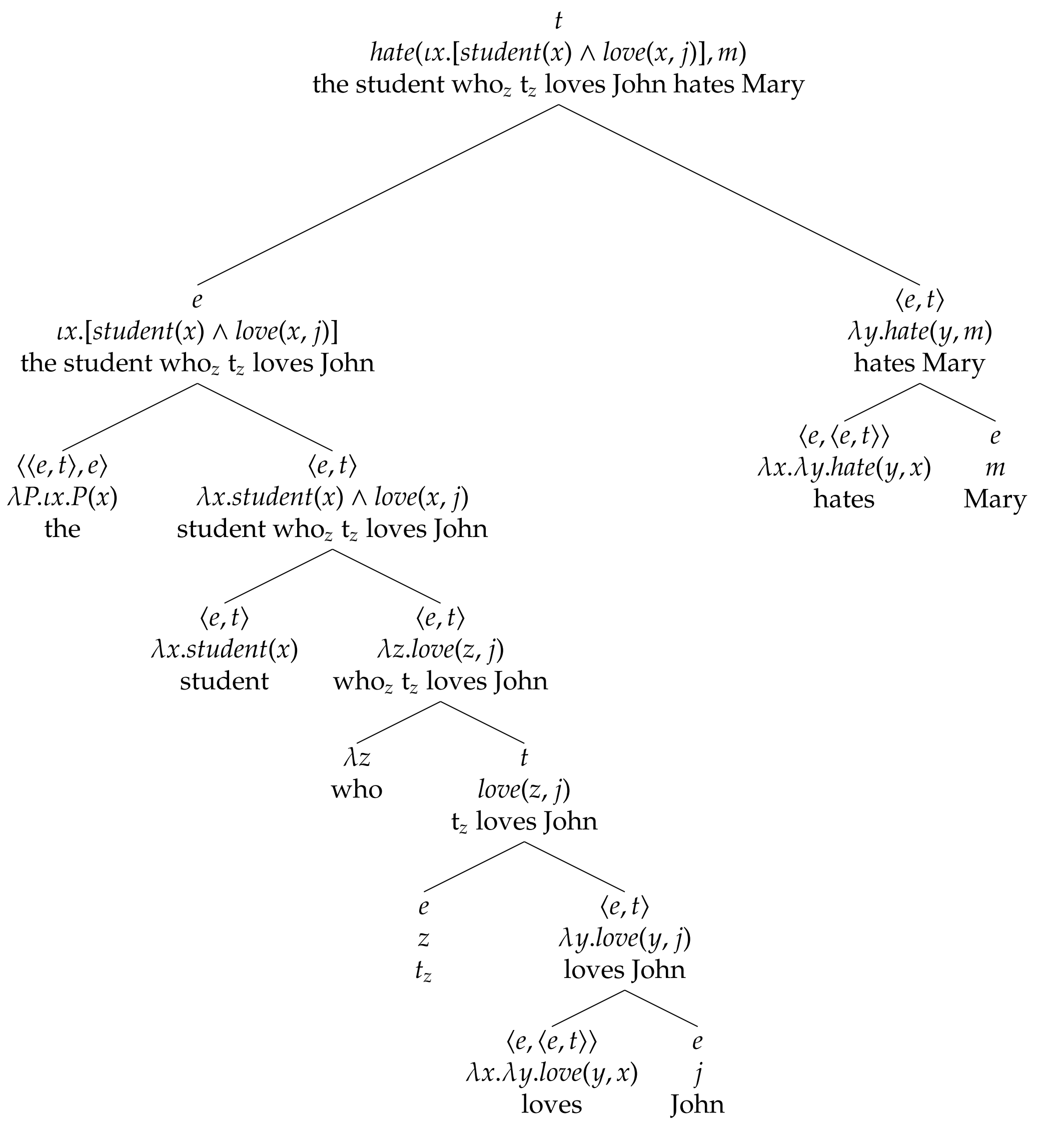
The trace in (74) is a variable bound by the lambda operator introduced by *who*. In logical form we use indices (e.g. z in (74)) to make clear which variable the trace corresponds to. Let us start by applying these assumptions to t_z *loves John*.

(75)



Given that the trace is a variable of type e, t_z *loves John* expresses a proposition. This is no ordinary proposition, however. The variable z in $love(z, j)$ is not bound, and so we have no way of interpreting the proposition, since we do not know what z refers to. (This is called an *open proposition*). Higher up in the logical form, however, the proposition is turned into a property by λ-binding the z variable. (We assume that the (empty) C-head has no semantic import.) The result is a full analysis as in (76).

(76)


t
$hate(\iota x.[student(x) \wedge love(x, j)], m)$
the student who$_z$ t_z loves John hates Mary
e
$\iota x.[student(x) \wedge love(x, j)]$
the student who$_z$ t_z loves John
$\langle e,t\rangle$
$\lambda y.hate(y,m)$
hates Mary
$\langle\langle e,t\rangle,e\rangle$
$\lambda P.\iota x.P(x)$
the
$\langle e,t\rangle$
$\lambda x.student(x) \wedge love(x, j)$
student who$_z$ t_z loves John
$\langle e,\langle e,t\rangle\rangle$
$\lambda x.\lambda y.hate(y,x)$
hates
e
m
Mary
$\langle e,t\rangle$
$\lambda x.student(x)$
student
$\langle e,t\rangle$
$\lambda z.love(z, j)$
who$_z$ t_z loves John
λz
who
t
$love(z, j)$
t_z loves John
e
z
t_z
$\langle e,t\rangle$
$\lambda y.love(y, j)$
loves John
$\langle e,\langle e,t\rangle\rangle$
$\lambda x.\lambda y.love(y,x)$
loves
e
j
John

10 Further reading

A good further introduction on types and lambdas can be found in two publicly available course handouts by Henriette de Swart:

`http://www.let.uu.nl/~Henriette.deSwart/personal/Classes/barcelona/typ.pdf`

&

`http://www.let.uu.nl/~Henriette.deSwart/personal/Classes/barcelona/lambda.pdf`

References

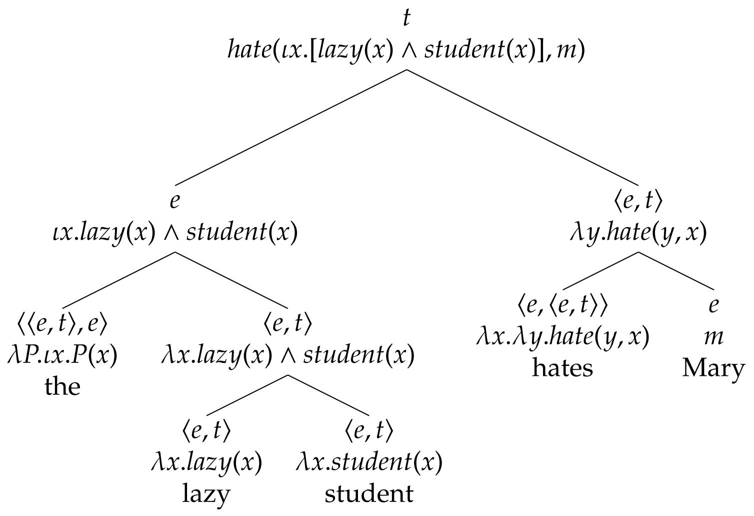
Heim, I. and A. Kratzer (1998). *Semantics in Generative Grammar*. Blackwell Publishing.

A Appendix: The horror of beta reduction

To the untrained eye, the implicit β-reduction steps in a semantic derivation will often appear to be an entirely random manipulation of variables. The only way to learn to fully grasp what is going on is to practice making such derivations and to make the β-reduction steps explicit. In this section, I will go through an example in great detail, and explain how beta reduction and variable choice works.

Consider the following example:

(77)



But what steps are implicit in here? We will go through the tree bottom-up.

First step: Function applying ⟦hates⟧ to ⟦Mary⟧.

(78) $\lambda x.\lambda y.hate(y, x)\ (m)$ function application
$= \lambda y.hate(y, m)$ beta reduction

Here, we have replaced the occurrences of x in the lambda-term $\lambda y.hate(y, x)$ with m. Note that the "(m)" in the first line of (78) is a case of function application. The "(y,x)" in $hate(y, x)$ might look the same, but this is not function application, this is specifying the arguments of the predicate *hate* at the predicate-logical level.

Notice that the order of the λ's is the reverse of the order of the argument of the predicate *hate*. This is because in a derivation the object is processed first. In $hate(y, x)$, it is the x variable that corresponds to the object (the theme of the hate relation), and so the first argument to be filled in is x.

Second step: Predicate modification of ⟦lazy⟧ and ⟦student⟧

(79)
$$\begin{tikzcd}[column sep=small] & \lambda x.lazy(x) \wedge student(x) \arrow[dl, dash] \arrow[dr, dash] & \\ \lambda x.student(x) & & \lambda x.lazy(x) \end{tikzcd}$$

Here, it is important to understand that the choice of the variable is immaterial. For instance, (80) is equivalent:

(80)
$$\begin{tikzcd}[column sep=small] & \lambda z.lazy(z) \wedge student(z) \arrow[dl, dash] \arrow[dr, dash] & \\ \lambda y.student(y) & & \lambda x.lazy(x) \end{tikzcd}$$

By predicate modification the λ-bound variables in the two $\langle e, t\rangle$ meanings are replaced by a single λ-binding and a conjunction of the predicates. The only way in which a choice of variable could be problematic is if there happens to be independent variable that is not lambda-bound. For instance, (81) is a good case of predicate modification, but (82) isn't, since the free variable z ends up bound.

(81)
$$\begin{tikzcd}[column sep=small] & \lambda x.student(x) \wedge hate(x, z) \arrow[dl, dash] \arrow[dr, dash] & \\ \lambda y.student(y) & & \lambda x.hate(x, z) \end{tikzcd}$$

(82)
$$\begin{tikzcd}[column sep=small] & \lambda z.student(z) \wedge hate(z, z) \arrow[dl, dash] \arrow[dr, dash] & \\ \lambda y.student(y) & & \lambda x.hate(x, z) \end{tikzcd}$$

Similarly, (83) is fine, but (84) isn't.

(83)
$$\begin{tikzcd}[column sep=small] & \lambda x.student(x) \wedge \exists z[hate(x, z)] \arrow[dl, dash] \arrow[dr, dash] & \\ \lambda y.student(y) & & \lambda x.\exists z[hate(x, z)] \end{tikzcd}$$

(84)
$$\begin{tikzcd}[column sep=small] & \lambda z.student(z) \wedge \exists z[hate(z, z)] \arrow[dl, dash] \arrow[dr, dash] & \\ \lambda y.student(y) & & \lambda x.\exists z[hate(x, z)] \end{tikzcd}$$

Third step: Function applying ⟦the⟧ to ⟦lazy student⟧

(85) $\lambda P.\iota x.P(x)\ (\lambda x.lazy(x) \wedge student(x))$ — function application
$\iota x.\lambda x.lazy(x) \wedge student(x)\ (x)$ — beta reduction + new function application
$\iota x.lazy(x) \wedge student(x)$ — beta reduction

This is probably one of the most confusing kind of reductions. In the first line in (85), we function apply the complex property corresponding to lazy student to the reference of the definite article. In the second line, we have replaced the P variable in the iota-term $\iota x.P(x)$ by $\lambda x.lazy(x) \wedge student(x)$. We are left with a new case of function application. To the right of ιx, the lambda-term $\lambda x.lazy(x) \wedge student(x)$ is function applied to x. To clarify, the structure of the form in the second line is a follows:

(86) $$\iota x.\ \overbrace{\underbrace{\lambda x.lazy(x) \wedge student(x)}_{\lambda\text{-term}}\ (x)}^{\text{scope of }\iota}$$

The last line of (85) is the beta reduction step. It replaces x in the scope of λx, that is $lazy(x) \wedge student(x)$ by x. This is the x that is bound by the iota operator.

It would have been easier to see what happens in this step had we have chosen a different variable for the lambda-binding in the property corresponding to *lazy student*. For instance:

(87) $\lambda P.\iota x.P(x)\ (\lambda y.lazy(y) \wedge student(y))$ — function application
$\iota x.\lambda y.lazy(y) \wedge student(y)\ (x)$ — beta reduction + new function application
$\iota x.lazy(x) \wedge student(x)$ — beta reduction

Here we see better what happens in the final step. The variable y in $lazy(y) \wedge student(y)$ is replaced by x. This x is now bound by the ι operator.

One final note on a potential source of confusion. In the reference of the definite description, it appears as if $P(x)$ is a normal predicate-term combination from predicate logic. It is not, however, this is a case of function application. This becomes clear from the fact that in the current example P is *not* a predicate from predicate logic, but rather a complex $\langle e, t\rangle$-type property.

Final step: Function applying ⟦hates Mary⟧ to ⟦the lazy student⟧

(88) $\lambda y.hate(y, m)\ (\iota x.lazy(x) \wedge student(x))$ — function application
$hate(\iota x.[lazy(x) \wedge student(x)], m)$ — beta reduction

In the beta reduction step we have replaced the y variable in $hate(y, m)$ by $\iota x.lazy(x) \wedge student(x)$.

B A quick example discussion of conjunction

(89) Sentence conjunction:
a. John hates Mary and Bill hates Sue.
b. $hate(j, m) \wedge hate(b, s)$

(90) Semantics of *and*
a. *and* is of type $\langle t, \langle t, t\rangle\rangle$
b. $\lambda p_t.\lambda p'_t.p \wedge p'$

(91) Beyond sentence conjunction:

a. John and Mary hate Sue.
b. John hates Sue and Mary.

(92) Early accounts:

a. John and Mary hate Sue $\rightsquigarrow$ John hates Sue and Mary hates Sue
b. John hates Sue and Mary $\rightsquigarrow$ John hates Sue and John hates Mary

(93) Problem with rewrite account:

Exactly two students hate Mary and Sue
(consider the case where Mary is hated by the students Bill, John and Harry, and Sue is hated by the students Bill, Harry and Larry.)

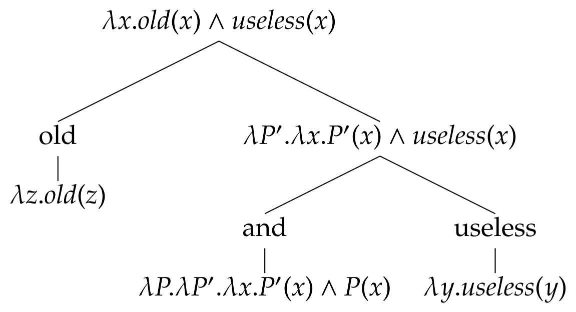
(94) Further cases:

a. John hates Mary and loves Sue.
b. old and useless

(95) $\langle e,t\rangle$-type conjunction

a. *and* is of type $\langle\langle e,t\rangle,\langle\langle e,t\rangle,\langle e,t\rangle\rangle\rangle$
b. $\lambda P_{\langle e,t\rangle}.\lambda P'_{\langle e,t\rangle}.\lambda x_e.P'(x) \wedge P(x)$

(96)



$\lambda x.old(x) \wedge useless(x)$
old
$\lambda z.old(z)$
$\lambda P'.\lambda x.P'(x) \wedge useless(x)$
and
$\lambda P.\lambda P'.\lambda x.P'(x) \wedge P(x)$
useless
$\lambda y.useless(y)$

(97) Draw a tree for example (94-a) and show the compositional meaning derivation in the same way as in (96)

(98) Develop a semantics for verb-conjunction by providing a lambda term for *and* of type $\langle\langle e,\langle e,t\rangle\rangle,\langle\langle e,\langle e,t\rangle\rangle,\langle e,\langle e,t\rangle\rangle\rangle\rangle$. Apply this semantics to account for the following example:

(i) John adores and admires Sue.

C A quick example discussion of negation

Negation in English often scopes over the complete sentence, even though it occurs at a relatively low surface position:

(99) a. John does not hate Mary.
b. $\neg[hate(j,m)]$

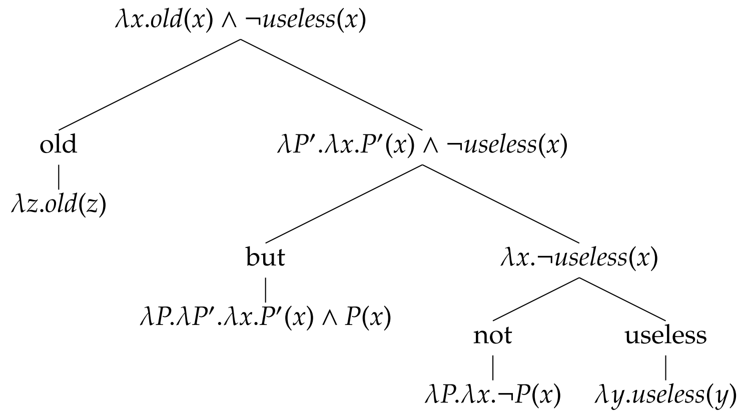
(100) Sentence negation

a. *not* is of type $\langle t, t\rangle$
b. $\lambda p_t.\neg p$

Not all negation scopes over the main clause:

(101) a. My car is old but not useless
b. $old(c) \wedge \neg useless(c)$

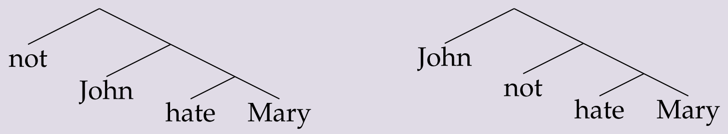
(102) Predicate negation:
a. *not* is of type $\langle\langle e,t\rangle,\langle e,t\rangle\rangle$
b. $\lambda P.\lambda x.\neg P(x)$

(103)

$\lambda x.old(x) \wedge \neg useless(x)$
old
$\lambda z.old(z)$
$\lambda P'.\lambda x.P'(x) \wedge \neg useless(x)$
but
$\lambda P.\lambda P'.\lambda x.P'(x) \wedge P(x)$
$\lambda x.\neg useless(x)$
not
$\lambda P.\lambda x.\neg P(x)$
useless
$\lambda y.useless(y)$

Take the following assumptions:

— *not* is ambiguous between (100-b) and (102-b)
— The auxiliary *does* is semantically vacuous

Now show that the two LFs in (104) predict just a single reading for (99-a):

(104)

not
John
hate
Mary
John
not
hate
Mary