Title	Chapter 5 : Functions, arguments and modifiers : Introduction to semantics for non-native speakers of English
Sub Title	
Author	Tancredi, Christopher
Publisher	慶應義塾大学言語文化研究所
Publication year	2025
Jtitle	慶應義塾大学言語文化研究所紀要 (Reports of the Keio Institute of Cultural and Linguistic Studies). No.56 (2025. 3) ,p.327- 340
JaLC DOI	10.14991/005.00000056-0327
Abstract	
Notes	研究ノート
Genre	Departmental Bulletin Paper
URL	https://koara.lib.keio.ac.jp/xoonips/modules/xoonips/detail.php?koara_id=AN00069467-00000056-0327

Reports of the Keio Institute of Cultural and Linguistic Studies 56 (2025), 327–340

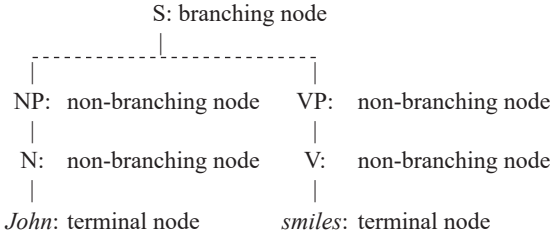
Chapter 5
Functions, Arguments and Modifiers

Introduction to Semantics for Non-native Speakers of English

Christopher Tancredi

The rules for translating English expressions into expressions of logic apply to syntactic trees. We are assuming that syntactic trees are at most binary branching. In Chapter 4 we considered three separate cases, summarized and shown below:

Terminal nodes: nodes at the bottom of the tree that do not dominate anything else
Non-branching nodes: nodes dominating a single daughter node
Branching nodes: nodes dominating two daughter nodes



The way to translate a node of a syntactic tree into an expression of logic depends on the type of node it is.

Terminal node: The translation is listed in the lexicon
Non-branching node: The translation is the same as the translation of its daughter node
Branching node: The translation is determined using Function Application

These rules are applied to our simple example of *John smiles* below.

Terminal nodes:

||*John*|| = john

||*smiles*|| = λx [SMILE (x)]

Non-branching nodes:

||NP|| = ||N|| = ||*John*|| (= john)

||VP|| = ||V|| = ||*smiles*|| (= λx [SMILE (x)])

Branching nodes:

||S|| = ||VP|| (||NP||)

= λx [SMILE (x)] (john)

= SMILE (john)

Vacuous Expressions

We have so far only translated very simple English sentences into logic. These sentences all contain a single verb with one or two arguments. Our next step is to translate sentences having adjectives or nouns as their main predicates.

Recall that adjectives translate as predicates in our logic, just like verbs do. For example, we have the following translations for adjectives:

||*happy*|| = λx [HAPPY (x)]

||*black*|| = λx [BLACK (x)]

||*proud*|| = λx [λy [PROUD (y,x)]]

Adjectives differ from verbs, however. While *John smiles* is an English sentence, *John happy* is not. To put *happy* together with *John* to get a sentence, we need the word *is*: *John is happy*. We need to determine what to do with this *is*.

Semantically, *is* plays an important role in the sentence *John is happy*: it marks the tense of the sentence. Other than that, however, its only role is to allow *happy* to combine with *John*. We can see this by looking at more complex examples like the following:

I consider John (to be) happy

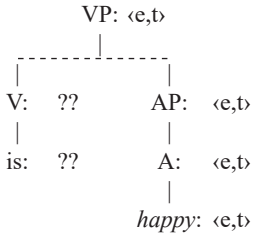
I saw John (being) happy

The words in parentheses do not affect the meanings of these sentences. The sentences mean the same with or without them. In these cases, then, the addition of *be* in the syntax has no effect on the semantics. We say that *be* is semantically empty, or **vacuous**. If we analyze *is* as *be* + present tense, *is* does not add any meaning to *John is happy* other than the meaning of its tense.

To analyze the sentence *John is happy*, we need an analysis of *is*. Ignoring tense, we want our analysis to have the following consequence:

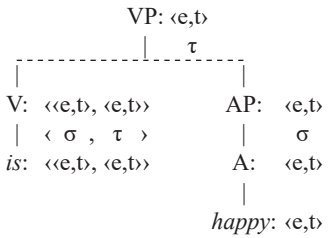
||is happy|| = ||happy||

How can we get this consequence? To answer that, let us first look at semantic types. If ||*is happy*|| is identical to ||*happy*||, then their semantic types are the same as well. Above we analyzed ||*happy*|| as λx [HAPPY (x)]. This is a function that takes an individual of type e and gives back a formula that is either true or false. Its semantic type is therefore ‹e,t›. We can use this information to calculate the semantic type of *is*. We start by putting all of the information that we know about types into the syntactic tree for *is happy*:



We then ask what the type of V has to be. The only rule we have for combining two expressions is the rule of Function Application. For this rule to apply, the types of the two expressions have to be of the form σ and ‹σ,τ›, for some types σ and τ. Given that AP is of type ‹e,t›, there are two classes of types for V that would allow AP and V to combine by Function Application. V could be of type e, or it could be of type ‹‹e,t›, τ› for some type τ. If V is of type e, however, the result of combining AP and V will

be of type t. That is, the result of combining *is* with *happy* will be a truth value. That is a wrong result. As marked in the tree, we want the result to be of type ‹e,t›. We can get that result by taking V to be of type ‹‹e,t›, ‹e,t››. This is illustrated below. There, the actual types ‹e,t› and ‹‹e,t›, ‹e,t›› are matched with the types σ, τ and ‹σ,τ› from the Function Application rule.



That ||*is*|| is of type ‹‹e,t›, ‹e,t›› means that ||*is*|| is a function that takes an ‹e,t›-type expression and gives back an ‹e,t›-type expression. From the requirement that ||*is happy*|| = ||*happy*||, we can determine that ||*is*|| is the following:

||*is*|| = λP [P], where P is of type ‹e,t›

Notice that we have specified the type of the variable P. This blocks ||*is*|| from combining with the wrong type of expression. From now on we will put information about the type of an argument on the variable introduced by λ, as follows:

$||is|| = \lambda P_{\langle e,t \rangle}\ [P]$

This says that ||*is*|| is a function that takes a 1-place predicate P as argument and gives back that same 1-place predicate P as its value. With this as our value for ||*is*||, we can now calculate the value of ||*is happy*|| as follows:

$$\begin{aligned} ||is\ happy|| &= ||is||\ (||happy||) \\ &= \lambda P_{\langle e,t \rangle}\ [P]\ (\lambda x_e\ [\text{HAPPY}\ (x)]) \\ &= \lambda x_e\ [\text{HAPPY}\ (x)] \end{aligned}$$

The calculation works in the same way as before. The only difference is in what gets

substituted. Before we always substituted an individual for a variable inside […]. This time, we substituted a function instead. In particular, we substitute the function $\underline{\lambda x_e\ [\text{HAPPY}\ (x)]}$ for the variable $\underline{P}$ in $[\underline{P}]$.

We decided in Chapter 4 to interpret nouns as predicates, just like adjectives and verbs. This means that we translate nouns as functions from (some number of) individuals to truth-value-denoting formulas. Examples are given below.

$\|\textit{dog}\| \quad = \lambda x_e\ [\text{DOG}\ (x)]$
$\|\textit{sister}\| \quad = \lambda x_e\ [\lambda y_e\ [\text{SISTER}\ (y,x)]]$

When used as the main predicate of a sentence, the noun *dog* requires the verb *is*. Nouns are like adjectives in this way. However, *dog* also often requires an **article**, either the **indefinite article** *a* or the **definite article** *the*. To say what kind of an animal Fido is, for example, we say *Fido is a dog*, not *Fido is dog*. To translate the sentence *Fido is a dog* into logic, we need to know how to translate the indefinite article *a*.

The word *a* almost certainly adds something to the meaning of *a dog*. To say something is *a dog* is to identify it as an individual animal. To say something is *dog*, without the word *a*, is typically to identify it as dog meat. These are very different things. Accounting for this difference, however, is an advanced topic. Still, it will be useful below to be able to use nouns as main predicates. This means we need to do something with *a*. We choose to analyze *a* as vacuous, just like *be*. This is not a serious proposal for the meaning of *a*. However, it makes it possible to analyze the sentence *Fido is a dog* and other sentences that have an indefinite noun phrase as their main predicate. This will be useful when we look at how nouns and adjectives combine in the next section.

The translation of *Fido is a dog* is given step by step below.

$\|\textit{Fido}\| \quad = \text{fido}$
$\|\textit{is}\| \quad = \lambda P_{\langle e,t\rangle}\ [P]$
$\|\textit{a}\| \quad = \lambda P_{\langle e,t\rangle}\ [P]$
$\|\textit{dog}\| \quad = \lambda x_e\ [\text{DOG}\ (x)]$

$$
\begin{aligned}
\|[\text{Fido [is [a dog]]}]\| &= \|[\text{is [a dog]}]\| (\|\text{Fido}\|) \\
&= \|\textit{is}\| (\|\textit{a dog}\|) (\|\textit{Fido}\|) \\
&= \|\textit{is}\| (\|\textit{a}\|) (\|\textit{dog}\|)) (\|\textit{Fido}\|) \\
&= \|\textit{is}\| (\lambda P_{\langle e,t\rangle} [P] (\lambda x_e [\text{DOG } (x)])) (\|\textit{Fido}\|) \\
&= \|\textit{is}\| (\lambda x_e [\text{DOG } (x)]) (\|\textit{Fido}\|) \\
&= \lambda P_{\langle e,t\rangle} [P] (\lambda x_e [\text{DOG } (x)]) (\|\textit{Fido}\|) \\
&= \lambda x_e [\text{DOG } (x)] (\|\textit{Fido}\|) \\
&= \lambda x_e [\text{DOG } (x)] (\text{fido}) \\
&= \text{DOG (fido)}
\end{aligned}
$$

Adjectives as Modifiers

The analysis of adjectives and nouns given above works for the examples we have looked at so far. However, the analysis fails to account for some simple observations. Consider the following three sentences:

A: *[Fido [is black]]*
B: *[Fido [is [a dog]]]*
C: *[Fido [is [a [black dog]]]]*

Intuitively, A and B together entail C: if A and B are true then so is C. Similarly, C entails A and B: if C is true, then so are A and B. We want these facts to follow from our analysis. However, our analysis does not yet have a way to analyze C.

The analyses of A and B are as follows:

Analysis of A:

$$
\begin{aligned}
\|\textit{Fido}\| &= \text{fido} \\
\|\textit{is}\| &= \lambda P_{\langle e,t\rangle} [P] \\
\|\textit{black}\| &= \lambda x_e [\text{BLACK}(x)] \\
\|\textit{is black}\| &= \|\textit{is}\| (\|\textit{black}\|) = \lambda P_{\langle e,t\rangle} [P] (\lambda x_e [\text{BLACK}(x)]) = \lambda x_e [\text{BLACK}(x)] \\
\|\textit{Fido is black}\| &= \|\textit{is black}\| (\|\textit{Fido}\|) = \lambda x_e [\text{BLACK}(x)] (\text{fido}) = \text{BLACK (fido)}
\end{aligned}
$$

Analysis of B:

$$
\begin{aligned}
\|\textit{a}\| &= \lambda P_{\langle e,t\rangle} [P] \\
\|\textit{dog}\| &= \lambda x_e [\text{DOG}(x)]
\end{aligned}
$$

$\|a\ dog\| = \|a\|(\|dog\|) = \lambda P_{\langle e,t\rangle}[P](\lambda x_e[DOG(x)]) = \lambda x_e[DOG(x)]$

$\|is\ a\ dog\| = \|is\|(\|a\ dog\|) = \lambda P_{\langle e,t\rangle}[P](\lambda x_e[DOG(x)]) = \lambda x_e[DOG(x)]$

$\|Fido\ is\ a\ dog\| = \|is\ a\ dog\|(\|Fido\|) = \lambda x_e[DOG(x)](fido) = DOG(fido)$

Trying to analyze C, however, we get stuck when we try to combine ||*black*|| with ||*dog*||.

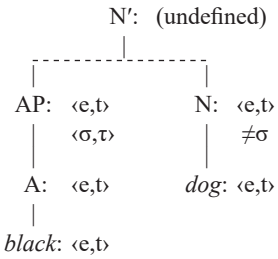
Analysis of C:

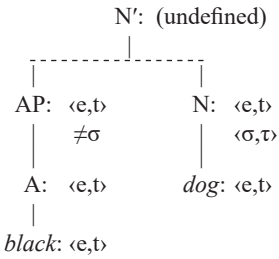
$\|black\| = \lambda x_e[BLACK(x)]$

$\|dog\| = \lambda x_e[DOG(x)]$

$\|black\ dog\| = ??$

Both *black* and *dog* translate into logic as ‹e,t›-type functions. Each requires something of type e to combine with, but neither is of type e. When the types of expressions fail to match up in this way, we say that their types **clash**. When two expressions have types that clash, they cannot be composed. We say that the result of trying to combine them is undefined.





Since it is intuitively possible to combine *black* and *dog* into *black dog*, this is a problem for our analysis.

There are two common ways of solving this problem. One is to change the types of the expressions so that they do not clash. The other is to add a new rule for combining expressions that makes it possible to combine two expressions of type ‹e,t›. We will look at both of these solutions below. Both of the solutions correctly predict that something that is a black dog is both black and a dog. They also correctly predict that

something that is black and a dog is a black dog.

Type-Raised Adjectives

The first solution continues to assume that ||*black*|| and ||*dog*|| combine by Function Application. To get this to work, either ||*black*|| or ||*dog*|| has to be a function that can take the other as argument. The normal choice is to take ||*black*|| to be the function and ||*dog*|| its argument. This choice makes it easier to analyze NPs like *a large*, *quiet*, *black dog* that have many adjectives modifying one noun.

To work out this solution, we take the type of *black dog* to be the same as the type of *dog*. Keeping the type of *dog* as ‹e,t›, the type of *black* is then ‹‹e,t›, ‹e,t››. Under these assumptions, the translation for *black* is:

$$||black|| = \lambda P_{\langle e,t\rangle} [\lambda y_e [\text{BLACK}(y) \,\&\, P(y)]]$$

Let us check that this has the right semantic type. We can read the semantic type off of the translation. $\lambda P_{\langle e,t\rangle}$ tells us that this is a function whose first argument is of type ‹e,t›. From this we can determine the type to be of the form:

‹‹e,t›, ___›

After taking an ‹e,t›-type argument, another function is returned. This new function begins with λy_e, telling us that it takes an e-type expression. From this we can add to our information about the type. It is of the form:

‹‹e,t›, ‹e,___ ››

Finally, after taking an e-type argument a formula is returned. A formula is either true or false. Its type is therefore type t. Adding in this last piece of information gives us the type of the whole:

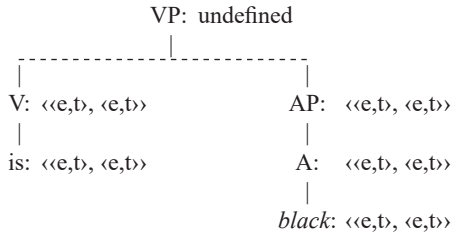
‹‹e,t›, ‹e,t››

Below we can see that this analysis of *black* gives us an acceptable interpretation for *Fido is a black dog*.

$$
\begin{aligned}
&\|\textit{dog}\| && = \lambda x_e\,[\mathrm{DOG}\,(x)] \\
&\|\textit{black dog}\| && = \|\textit{black}\|\,(\|\textit{dog}\|) && \text{(By Function Application)} \\
& && = \lambda P_{\langle e,t\rangle}\,[\lambda y_e\,[\mathrm{BLACK}\,(y)\ \&\ P\,(y)]]\,(\lambda x_e\,[\mathrm{DOG}\,(x)]) \\
& && = \lambda y_e\,[\mathrm{BLACK}\,(y)\ \&\ \lambda x_e\,[\mathrm{DOG}\,(x)]\,(y)] \\
& && = \lambda y_e\,[\mathrm{BLACK}\,(y)\ \&\ \mathrm{DOG}\,(y)] \\
&\|\textit{is a black dog}\| && = \|\textit{a black dog}\| = \|\textit{black dog}\| \\
&\|\textit{Fido is a black dog}\| && = \|\textit{is a black dog}\|\,(\|\textit{Fido}\|) \\
& && = \lambda y_e\,[\mathrm{BLACK}\,(y)\ \&\ \mathrm{DOG}\,(y)]\,(\mathrm{fido}) \\
& && = \mathrm{BLACK}\,(\mathrm{fido})\ \&\ \mathrm{DOG}\,(\mathrm{fido})
\end{aligned}
$$

According to this analysis, *Fido is a black dog* is true just in case Fido is black and Fido is a dog.

Note that the new definition for ||*black*|| only works in the sentence *Fido is a black dog*. If we try to use it in the sentence *Fido is black* we get a type clash. This is because the type of *is* is ‹‹e,t›, ‹e,t››, just like the type of the newly analyzed *black*, and two expressions of the same type can never combine by Function Application.



We can analyze both sentences under this approach if we take *black* to be ambiguous. In particular, this will work if *black* has an ‹‹e,t›, ‹e,t››-type translation in the sentence *Fido is a black dog* and an ‹e,t›-type translation in the sentence *Fido is black*. This gives us:

$$
\begin{aligned}
&\|\textit{black}_1\| = \lambda x_e\,[\mathrm{BLACK}\,(x)] \\
&\|\textit{black}_2\| = \lambda P_{\langle e,t\rangle}\,[\lambda y_e\,[\mathrm{BLACK}\,(y)\ \&\ P\,(y)]]
\end{aligned}
$$

Note that this ambiguity is like the ambiguity of *eat* that we looked at in Chapter 3. $Black_1$ and $black_2$ differ syntactically, and because of this they have different semantic types. However, their logical translations are both based on the same predicate BLACK. Because of this, the analysis predicts that sentences using these two lexical items can entail one another. Consider *Fido is a* $black_2$ *dog* and *Fido is* $black_1$. The first sentence translates into logic as BLACK (fido) & DOG (fido). The second sentence translates into logic as BLACK (fido). Since in logic p & q entails p for any formulas p and q, the first sentence is predicted to entail the second. This entailment is just what we observe.

If the English word *black* is formally ambiguous between $black_1$ and $black_2$, then a similar ambiguity will be required for other adjectives as well. After all, *Fido is a nice dog* entails *Fido is nice*, and in many cases a sentence of the form *Fido is a(n) ADJ dog* entails *Fido is ADJ* when both occurrences of *ADJ* are replaced by the same adjective. To account for this, these adjectives need to come in two forms: ADJ_1 of type ‹e,t›, and ADJ_2 of type ‹‹e,t›, ‹e,t››.

For any English adjective *ADJ*, the logical translations of these two representations ADJ_1 and ADJ_2 have to be based on the same logical predicate. If ADJ_1 and ADJ_2 are simply listed as separate words in the lexicon, there is no way to guarantee this result. However, we can guarantee the result if one of the translations is based on the other. To see how, consider the case of *black*. We see below that $||black_2||$ can be defined in terms of $||black_1||$.

$$
\begin{aligned}
||black_1|| &= \lambda x_e\,[\text{BLACK}\,(x)] \\
||black_2|| &= \lambda P_{\langle e,t\rangle}\,[\lambda y_e\,[||black_1||\,(y)\ \&\ P\,(y)]] \\
&= \lambda P_{\langle e,t\rangle}\,[\lambda y_e\,[\lambda x_e\,[\text{BLACK}\,(x)]\,(y)\ \&\ P\,(y)]] \\
&= \lambda P_{\langle e,t\rangle}\,[\lambda y_e\,[\text{BLACK}\,(y)\ \&\ P\,(y)]]
\end{aligned}
$$

By defining $||black_2||$ in terms of $||black_1||$ in this way, we can be certain that both translations are based on the same logical predicate, here BLACK. The same will hold for other adjectives as well.

We generalize this relation between ‹e,t›- and ‹‹e,t›, ‹e,t››-type translations for adjectives as follows:

For any adjective *ADJ* of type ‹e,t›, there is an ‹‹e,t›, ‹e,t››-type adjective ADJ_2 having the same form as *ADJ*, where:

$||ADJ_2|| = \lambda P_{\langle e,t\rangle} [\lambda y_e [||ADJ|| (y) \& P (y)]]$

This is a lexical rule that creates one lexical item out of another. This rule does not add any new logical predicates to the translation but only changes how an expression can combine with other expressions. We call rules like this **type-shifting rules**. When the new expression has a more complicated, or higher, semantic type than the expression it is based on, the rule is a **type-raising rule**. The rule defining $||ADJ_2||$ based on $||ADJ||$ is a type-raising rule. When the new expression has a less complicated, or lower, semantic type, the rule is a **type-lowering rule**.

Predicate Modification

The solution we just looked at makes a trade-off. It keeps Function Application as the only rule that combines two expressions. However, because of that, it has to treat some expressions as ambiguous. The next solution makes the opposite trade-off. It treats adjectives and nouns as unambiguous. However, to do so it adds a new rule for combining expressions. This new rule combines two 1-place predicates into a new 1-place predicate. That is, it combines two ‹e,t›-type expressions to make a new, complex ‹e,t›-type expression. This rule is called Predicate Modification:

Predicate Modification: $||P_{\langle e,t\rangle}\ Q_{\langle e,t\rangle}|| = \lambda z_e [||P|| (z) \& ||Q|| (z)]$

Formally, this rule takes two 1-place predicates *P* and *Q*, applies each of their translations to a variable *z*, conjoins the results with &, and binds *z* with λz_e. The result is a 1-place predicate that is true of an individual if and only if the translations of both *P* and *Q* are true of that individual.

It is important to see that Predicate Modification is not just another way of doing Function Application. When *P* and *Q* are each predicates of type ‹e,t›, then $||P\ Q||$ does not equal either $||P||$ ($||Q||$) or $||Q||$ ($||P||$). The rule of Predicate Modification is a rule that is used *instead of* the rule of Function Application.

$||black|| = \lambda x_e [BLACK(x)]$

$||dog||$ $= \lambda x_e$ [DOG(x)]

$||black\ dog||$ $= \lambda z_e$ [$||black||$ (z) & $||dog||$ (z)] (By Predicate Modification)

$= \lambda z_e$ [λx_e [BLACK(x)] (z) & λx_e [DOG(x)] (z)]

$= \lambda z_e$ [BLACK (z) & DOG (z)]

$||is\ a\ black\ dog||$ $= ||a\ black\ dog|| = ||black\ dog||$

$||Fido\ is\ a\ black\ dog||$ $= ||is\ a\ black\ dog||$ ($||Fido||$)

$= \lambda x_e$ [BLACK (x) & DOG (x)] (fido)

= BLACK (fido) & DOG (fido)

The result of the Predicate Modification analysis of the sentence *Fido is a black dog* is the same as the result of the type-raising analysis. Both translate the sentence into the same logical formula: BLACK (fido) & DOG (fido). Therefore, the same entailment relations are accounted for under the two analyses.

Relative Clauses

A relative clause is a **clause**, that is a sentence-like expression, used to modify a noun. As a noun modifier, it functions in the same way that an adjective does. The italicized expressions below are all relative clauses.

a dog *that is black*
a man *who feeds Fido*
a toy *Fido plays with*

A relative clause is formed from a clause in which a predicate is missing a syntactic argument. In the relative clause *that is black* above, the missing argument is the subject of *black*. In *who feeds Fido*, it is the subject of *feeds*. In *Fido plays with*, it is the object of *with*. A relative clause can optionally start with the word *that*, or with a wh-word like *who*, *which*, *why*, *when*, *where*, *how*, and in some dialects of English *what*. The choice of the wh-word depends on the NP that the relative clause is modifying.

The formal rules for interpreting relative clauses are too complicated for an introductory textbook. However, the results of applying these rules are easy to describe. Just like there were two ways of analyzing adjectives, there are two ways of analyzing relative clauses. On one analysis, a relative clause is a 1-place predicate, of

type ‹e,t›. It combines with a noun like *dog* using the rule of Predicate Modification. On the other analysis, a relative clause is a function of type ‹‹e,t›, ‹e,t››. It combines with a noun like *dog* using Function Application.

Consider the 1-place predicate analysis of relative clauses first. On that analysis, the logical translation of a relative clause is a lambda function of the form λx_e […x…]. The formula inside the square brackets is the interpretation of the clause following the optional *that* or wh-word, with x filling in as the translation of the missing syntactic argument. The optional words *that*, *who*, … at the beginning of the relative clause are ignored. For the relative clauses above, the resulting translations are the following:

$$\begin{aligned}
&||\textit{that is black}|| &&= \lambda x_e\ [\text{BLACK}\ (x)]\\
&||\textit{who feeds Fido}|| &&= \lambda x_e\ [\text{FEED}\ (x, \text{fido})]\\
&||\textit{Fido plays with}|| &&= \lambda x_e\ [\text{PLAY-WITH}\ (\text{fido}, x)]
\end{aligned}$$

Under this analysis, the sentence *Fido is a dog that is black* gets the same logical translation as the sentence *Fido is a black dog*:

$$\begin{aligned}
&||\textit{dog}|| &&= \lambda x_e\ [\text{DOG}\ (x)]\\
&||\textit{that is black}|| &&= \lambda x_e\ [\text{BLACK}\ (x)]\\
&||\textit{dog that is black}|| &&= \lambda z_e\ [||\textit{dog}||\ (z)\ \&\ ||\textit{that is black}||\ (z)]\\
&&&\qquad\qquad\qquad\qquad\text{(by Predicate Modification)}\\
&&&= \lambda z_e\ [\lambda x_e\ [\text{DOG}\ (x)]\ (z)\ \&\ \lambda x_e\ [\text{BLACK}\ (x)]\ (z)]\\
&&&= \lambda z_e\ [\text{DOG}\ (z)\ \&\ \text{BLACK}\ (z)]\\
&||\text{a}|| &&= \lambda P_{\langle e,t\rangle}\ [P]\\
&||\text{a dog that is black}|| &&= ||\textit{a}||\ (||\textit{dog that is black}||)\\
&&&= \lambda P_{\langle e,t\rangle}\ [P]\ (\lambda z_e\ [\text{DOG}\ (z)\ \&\ \text{BLACK}\ (z)])\\
&&&= \lambda z_e\ [\text{DOG}\ (z)\ \&\ \text{BLACK}\ (z)]\\
&||\text{is}|| &&= \lambda P_{\langle e,t\rangle}\ [P]\\
&||\text{is a dog that is black}|| &&= ||\textit{is}||\ (||\textit{a dog that is black}||)\\
&&&= \lambda P_{\langle e,t\rangle}\ [P]\ (\lambda z_e\ [\text{DOG}\ (z)\ \&\ \text{BLACK}\ (z)])\\
&&&= \lambda z_e\ [\text{DOG}\ (z)\ \&\ \text{BLACK}\ (z)]\\
&||\text{Fido is a dog that is black}|| &&= ||\textit{is a dog that is black}||\ (||\textit{Fido}||)\\
&&&= \lambda z_e\ [\text{DOG}\ (z)\ \&\ \text{BLACK}\ (z)]\ (\text{fido})\\
&&&= \text{DOG}\ (\text{fido})\ \&\ \text{BLACK}\ (\text{fido})
\end{aligned}$$

The second analysis of relative clauses is a type-raised version of the first analysis. It is what you get by applying the type-raising rule we introduced earlier to relative clauses. The general form of the logical translation of a relative clause on this analysis is $\underline{\lambda P_{\langle e,t\rangle}\,[\lambda x_e\,[\ldots x\ldots\ \&\ P(x)]}$, where $\underline{\lambda x_e\,[\ldots x\ldots}$ is the same as on the earlier analysis. This analysis gives us the same results as the first analysis. That is, the result of combining the relative clause *that is black* with the noun *dog* ends up the same in the two analyses.

$$
\begin{aligned}
&\|\textit{dog}\| && = \lambda x_e\,[\mathrm{DOG}\,(x)]\\
&\|\textit{that is black}\| && = \lambda P_{\langle e,t\rangle}\,[\lambda y_e\,[\mathrm{BLACK}\,(y)\ \&\ P\,(y)]]\\
&\|\textit{dog that is black}\| && = \|\textit{that is black}\|\,(\|\textit{dog}\|) \qquad \text{(by Function Application)}\\
& && = \lambda P_{\langle e,t\rangle}\,[\lambda y_e\,[\mathrm{BLACK}\,(y)\ \&\ P\,(y)]]\,(\lambda x_e\,[\mathrm{DOG}\,(x)])\\
& && = \lambda y_e\,[\mathrm{BLACK}\,(y)\ \&\ \lambda x_e\,[\mathrm{DOG}\,(x)]\,(y)]\\
& && = \lambda y_e\,[\mathrm{BLACK}\,(y)\ \&\ \mathrm{DOG}\,(y)]
\end{aligned}
$$

The translation here is the same as the one from the first analysis. The only difference is in the variables that are used: $\underline{z}$ in the first analysis and $\underline{y}$ in the second. Since these variables occur in the same places, this difference has no semantic effect. It is like the difference between *f(x) = x+3* and *f(y) = y+3*. These are just two slightly different ways of describing the exact same function.