Title	Chapter 4 : Functions, arguments, and semantic types : 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	2024
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.55 (2024. 3) ,p.261- 275
JaLC DOI	10.14991/005.00000055-0261
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=AN00069467-00000055-0261

Reports of the Keio Institute of Cultural and Linguistic Studies 55 (2024), 261–275

Chapter 4
Functions, Arguments, and Semantic Types
Introduction to Semantics for Non-native Speakers of English

Christopher Tancredi

4.1 Introduction

In this textbook we are analyzing English expressions in two steps. First, we translate them into expressions of logic and then we assign a denotation to the logical expressions. In Chapter 2 we introduced a simplified logic, propositional logic, that took its basic expressions to be propositions. This made it possible to analyze *and*, *or*, *if* and *not* as operators over propositions. In Chapter 3 we added an analysis of sentences made of predicates and their arguments. To do so, we expanded our logic to include formulas made up of an *n*-place predicate combined with *n* terms. We gave set theoretic denotations for expressions of the logic, and we made a rule that gave us the truth-value of a formula based on the denotations of a predicate and its arguments. We then made some decisions about what the logical translations of English sentences should look like. However, we did not show how to derive those translations. In this chapter we fill in that missing piece. That is, we look into the box below.

English expression	**⇒ (translates as)**	**Logical expression**	**⇒ (denotes)**	**Set theoretic value**
John smiles		SMILE (john)		True if $j \in$ SMILE′ False otherwise
John sees Mary		SEE (john, mary)		True if $\langle j,m \rangle \in$ SEE′ False otherwise

4.2 Compositional Translation

Until now we have been translating English sentences as predicate logic formulas in

one step. We looked at the sentences and gave logical formulas that give us their truth conditions. If our only goal is to give truth conditions for sentences, this approach works well. However, sentences are not the only expressions that have meanings. Words like *John* and *sees* as well as phrases like *sees Mary* and *in the park* have meanings as well. Simply translating sentences into complete formulas does not tell us what these meanings are.

Our goal in semantics is to give meanings to all expressions of a language. However, we want to give those meanings in a special way. We want to **define** the meanings only of individual words. The meanings of complex expressions we want to build up, or **derive**, from the meanings of the words they are composed of. This is the basic idea behind compositionality.

We take the translation of a name like *John* to be very simple. An English name translates as a logical name: *John* translates as john. For a predicate like *smiles*, however, the translation is more complex. We cannot translate *smiles* as SMILE. Translating *John* as john and *smiles* as SMILE does not tell you how to put SMILE and john together in the logic. We said in Chapter 3 that a predicate like *smiles* acts like instructions for making a picture. It takes an individual and gives back a picture of that individual smiling. Translating *smiles* as SMILE misses these instructions.

These instructions we **formalize**, or analyze, as a **function**. The logical translation of *smiles* is a function that takes an individual as an argument and gives back a formula. This idea of a function is the same one that is used in math. Just like "+ 3" can be expressed as the function f below, the meaning of *smiles* can be expressed as the function g:

f(x) = x+3	f(7) = 7+3
g(x) = SMILE (x)	g(john) = SMILE (john)

The normal way we describe functions in semantics, however, is different. We use **lambda calculus**. The lambda calculus versions of f and g are given below:

f	⇒	λx [x+3]	λx [x + 3] (7) = 7+3
g	⇒	λx [SMILE (x)]	λx [SMILE (x)] (john) = SMILE (john)

In lambda calculus, a function is made up of λ, a variable (here x), and a bracketed expression […] that contains **occurrences**, or instances, of the variable. You can think of λx as instructions to find something that can substitute for the occurrences of x inside the brackets. That something is the argument of the function. It is found in the parentheses (…) following the lambda function. In the math example, the argument of the function *λx[x+3]* is the number 7. In the linguistic example, the argument of the function *λx[SMILE (x)]* is the logical name john. In both cases, the argument substitutes for the occurrence of x inside the square brackets to give the result: 7+3 for the math example, and SMILE (john) for the linguistic example. This process of substituting the argument for the variable is called **lambda conversion**. This process also eliminates the occurrence of *λx* and the square brackets.

The last piece we need to formalize translation from English to logic is a translation function. We use ||…|| as our translation function, and we read || E || as *the logical translation of E*.

||*English expression*|| = logical expression

For simple expressions, their logical translation is listed in the lexicon. This includes the translations of *John* and of *smiles* below:

||*John*|| = john
||*smiles*|| = λx [SMILE (x)]

For complex expressions, their translation is given by function application, defined as follows:

Function Application
For an expression E of the form $[_E$ A B] or $[_E$ B A],
||E|| = ||A|| (||B||)
or
||E|| = ||B|| (||A||)

The choice depends on what kinds of expressions A and B are. If the translation of A

is a function that can apply to the translation of B, then the first option is chosen. In the opposite case, the second option is chosen.

We can now see how to translate complex expressions like the sentence *John smiled* into logic. Its translation goes like this:

||*John smiles*||	= ||*smiles*|| (||*John*||)	[by function application]
	= λx [SMILE (x)] (john)	[by lexical translation]
	= SMILE (john)	[by lambda conversion]

In the first line, the translation of the sentence is broken into the translations of the two pieces that make it up, *smiles* and *John*. Since the translation of *smiles* is a function that can apply to the translation of *John*, we choose the second option from the rule of Function Application. The next line replaces ||*smiles*|| with the logical translation of *smiles*, namely λx[SMILE (x)], and it replaces ||*John*|| with the logical translation of *John*, namely john. In the third line, the argument john has been substituted for the occurrence of x inside the square brackets.

In the sentence *John smiled*, the two parts that combine into the sentence are both words. This is not always the case for other sentences, though. Consider the sentence *John sees Mary*. Syntax tells us that this sentence can be broken into two parts as well: *John*, and *sees Mary*. We can get the translation of *John* from the lexicon, as before. However, we cannot get the translation of *sees Mary* that way. Rather, we have to analyze this expression using Function Application. That is, we have to apply the translation of *sees* to the translation of *Mary* to get the translation of *sees Mary*.

What is the translation of *sees*? We can answer this question starting with the following five assumptions:

||*John*||	= john
||*Mary*||	= mary
||*John sees Mary*||	= SEE (john, mary)

John sees Mary is structured as *[John [sees Mary]]*

The translation is given by Function Application

To combine two things by Function Application, one of them must be a function and the other its argument. The translation of *John*, namely john, is not a function. In order to combine john with the translation of *sees Mary*, it follows that *sees Mary* must be a function. This gives us the following:

$$\begin{aligned} \|[\textit{John [sees Mary]}]\| &= \|[\textit{sees Mary}]\| (\|\textit{John}\|) \\ &= \lambda x [\ldots] (\text{john}) \end{aligned}$$

We also know that the result of applying the function λx […] to john must be SEE (john, mary). Applying λx […] to john results in john replacing any occurrences of x in […]. From this, we can guess what […] must be. There are two possibilities. Either […] contains a single occurrence of x that gets substituted by john, or it contains no occurrences of x.

$$\begin{aligned} &\|[\textit{sees Mary}]| (\|\textit{John}\|)| = \lambda x [\ldots] (\text{john}) = \text{SEE (john, mary)} \\ \Rightarrow\ &\lambda x [\ldots] = \lambda x [\text{SEE (x, mary)}] \\ &\text{OR} \\ &\lambda x [\ldots] = \lambda x [\text{SEE(john, mary)}] \end{aligned}$$

Of these two options, the first is clearly a better option for the translation of *sees Mary*. The expression *sees Mary* intuitively does not contain any information about John, so its translation should not either.

$$\Rightarrow \quad \|[\textit{sees Mary}]\| = \lambda x [\text{SEE (x, mary)}]$$

We can use the same process to determine the translation of *sees*. The translation of *sees Mary* combines the translation of *sees* with the translation of *Mary* by Function Application. The translation of *Mary* is mary, which is not a function. This means that the translation of *sees* must be a function. Also, we just determined that the translation of *sees Mary* is λx [SEE (x, mary)]. This has to be the result of applying the translation of *sees* to the translation of *Mary*. This gives us the following:

$$\begin{aligned} \|[\textit{sees Mary}]\| &= \lambda x [\text{SEE (x, mary)}] \\ &= \|\textit{sees}\| (\|\textit{Mary}\|) \\ &= \lambda y [\ldots] (\text{mary}) \end{aligned}$$

λx [SEE (x, mary)] results from substituting mary for y in …

⇒ λy […] = λy [λx [SEE (x, y)]]

⇒ ||*sees*|| = λy [λx [SEE (x, y)]]

We can show that this translation of *sees* works by giving a full derivation of *John sees Mary*. The derivation can be done from the bottom up or from the top down. Both are shown below.

Shared assumptions:

||*John*||	= john
||*Mary*||	= mary
||*sees*||	= λy [λx [SEE (x, y)]]

Bottom up:

||*[sees Mary]*||	= ||*sees*|| (||*Mary*||)
	= λy [λx [SEE (x, y)]] (mary)
	= λx [SEE (x, mary)]
||*[John [sees Mary]]*||	= ||*[sees Mary]*|| (||*John*||)
	= λx [SEE (x, mary)] (john)
	= SEE (john, mary)

Top down:

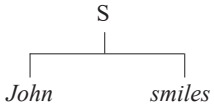
||*[John [sees Mary]]*||	= ||*[sees Mary]*|| (||*John*||)
	= ||*sees*|| (||*Mary*||) (||*John*||)
	= λy [λx [SEE (x, y)]] (mary) (john)
	= λx [SEE (x, mary)] (john)
	= SEE (john, mary)

4.3 Syntax

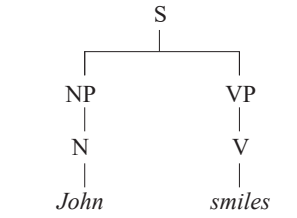
To calculate the meaning of *John sees Mary*, we assumed that *sees Mary* is an expression having *sees* and *Mary* as parts. Because of this structure, we combined the translation of *sees* with the translation of *Mary*. We did not combine the translation of *sees* directly with the translation of *John* because *John sees* is not an expression. This shows an important property of our semantics: it depends on structure. In particular,

the translation of an English expression into logic depends on the structure of the English expression.

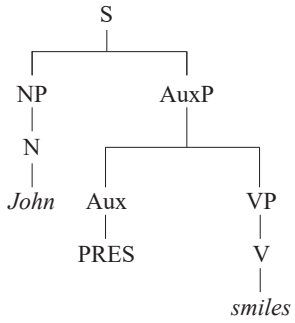
Until now we have only used very simple structures. In the study of syntax, however, more complex structures are used. Consider the sentence *John smiles*. We have been assuming a structure like the following:



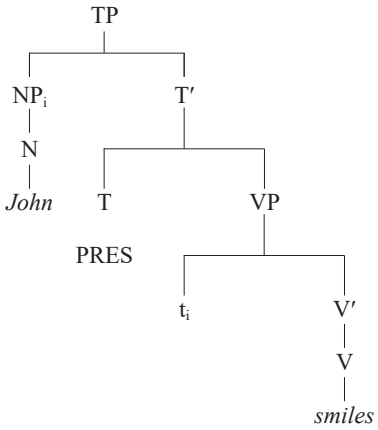
This works for our purposes here. However, for many syntacticians, this is not a realistic structure. Syntacticians have made many more complex proposals for the structure of *John smiles*, including the following. The sentences are represented using trees and using labeled brackets. The two representations are equivalent: they each represent the exact same structural information.



[S [NP [N *John*]] [VP [V *smiles*]]]



[S [NP [N *John*]] [AuxP [Aux PRES] [VP [V *smiles*]]]]



[$_{TP}$ [$_{NP_i}$ [$_N$ *John*]] [$_{T'}$ [$_T$ PRES] [$_{VP}$ t$_i$ [$_{V'}$ [$_V$ *smiles*]]]]]

It is not the job of semantics to decide which of these structures is correct. It is the job of semantics to interpret the structures it gets, though. Our semantics does not yet have the tools it needs to interpret all of these structures. Two things are missing. First, we do not have any rules for translating structures like the following into predicate logic:

X
|
Y

[X Y]

To translate these structures, we use the following rule:

Translation of Non-branching Structures

If a node X dominates a single daughter node Y, then the logical translation of X is equal to the logical translation of Y:

$\| [_X\ Y] \| = \|Y\|$

OR

$\left\| \begin{array}{c} X \\ | \\ Y \end{array} \right\| = \|Y\|$

Second, we do not have any rules for interpreting movement structures. In the last of the representations above, NP_i starts in the position where t_i is and moves to the

position it is seen in. This movement leaves t_i behind as a **trace**. We will not give rules for interpreting movement structures in this book. That is a topic for a more advanced course. Because of this, we only use structures that do not involve movement.

Even without movement, we have many structures to choose from. Do we analyze the tense in *John smiles* as something separate from the verb, for example? In general, we will only use complex structures when we are ready to interpret them. In the case of tense, we will not interpret tense until Chapter 10, so we will not include tense as a separate expression until then.

4.4 Types

Different expressions typically have different denotations. However, we can group expressions together based on the *kinds* of denotation they have. Expressions that have the same kind of denotation are of the same **semantic type**. For example, the expressions *Mary*, *John*, *Bill* and *Sue* are all of the same semantic type. Each denotes an individual. The sentences *John smiled* and *John didn't smile* are also of the same semantic type. Each denotes a truth value. The predicates *smile* and *sit* are of the same semantic type as well. Each denotes a function from individuals to truth-values.

We define two **basic** semantic types: **type e**, for **entities** (objects, individuals), and **type t** for truth values. Names like *John*, *Tokyo*, etc. are of type e: they denote entities of some sort, a person in the case of *John* and a city in the case of *Tokyo*. Sentences like *John smiles* are of type t: they denote a truth value, True if the sentence is true and False if the sentence is false.

In addition to the basic types, there are also **complex** types. These have the form $\langle\sigma,\tau\rangle$, where both σ (sigma) and τ (tau) are types. Something of type $\langle\sigma,\tau\rangle$ is a function from σ-type things to τ-type things. A one-place predicate like *smile* is of type $\langle e,t\rangle$, with *e* playing the role of σ and *t* playing the role of τ. It is a function from e-type things (entities) to t-type things (truth values). We have more complex types as well. The two-place predicate *see* that we analyzed above is of type $\langle e,\langle e,t\rangle\rangle$. It is a function from e-type things to $\langle e,t\rangle$-type things.

We define types **recursively** as follows:

Type	**Denotation**
e is a type.	entity (object, individual)
t is a type.	truth-value
If σ and τ are types, ⟨σ,τ⟩ is a type.	function from σ-type denotations to τ-type denotations

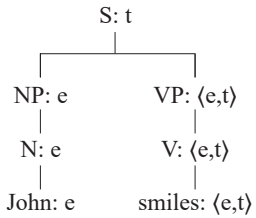
This definition gives us an infinite number of types. This comes from the recursive step at the end that defines new types based on other types. In addition to the types we have already seen, this definition also gives us the following:

⟨e, ⟨e, ⟨e, t⟩⟩⟩	[*show*]
⟨⟨e, t⟩, e⟩	[*the*]
⟨⟨e, t⟩, ⟨e, t⟩⟩	[*very*]
⟨⟨e, t⟩, t⟩	[*everyone*]
⟨t, e⟩	?
⟨e, ⟨t, e⟩⟩	?
⟨⟨e, t⟩, ⟨⟨⟨t, ⟨e, t⟩⟩, e⟩, ⟨e, e⟩⟩⟩	?

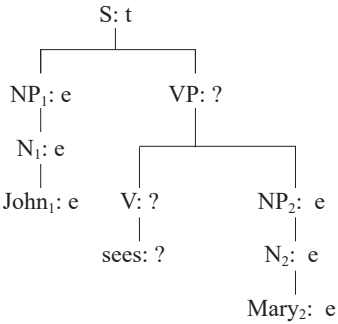
Only a very small number of types are used in semantics. This includes the top four types shown above, which are the types of the words *show*, *the*, *very* and *everyone*. Most types are not used in semantics, however. As far as I know, for example, the bottom three types are never used in semantics.

Types can help you to see whether two expressions can combine into one. The rule of Function Application tells you to combine the translations of two expressions A and B by applying ||A|| to ||B|| or by applying ||B|| to ||A||. The choice depends on what kinds of expressions A and B are. With semantic types, we can make it clear what this means. If, for some values of σ and τ, A is of type ⟨σ,τ⟩ and B is of type σ, then we have to choose the first option: ||[A B]|| = ||A|| (||B||). If, for some types σ and τ, A is of type σ and B is of type ⟨σ,τ⟩, then we have to choose the second option: ||[A B]|| = ||B|| (||A||). If neither case holds, the two expressions cannot combine by Function Application.

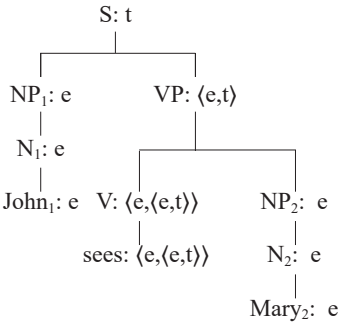
We can indicate types directly in the syntax as follows:



A two-place predicate like *see* combines with two noun phrases to produce a sentence that can be true or false. If we assume that branching is **binary**, that is that a node in a syntactic tree will never have more than 2 daughter nodes, *see* has to combine with one noun phrase first and then another. Knowing that the two expressions it combines with can be names, we can calculate the predicate's type as follows. We start with the information we know: The sentence is of type t since it can be true or false, and the two arguments *John* and *Mary* are type e since they denote individuals. This gives us the following types:



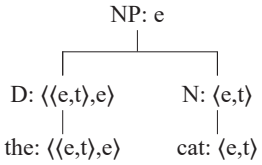
If we assume that VP and NP_1 combine by Function Application, we can determine the type for VP. It must be a function that combines with an e-type expression to produce a t-type expression. That is, its type must be ⟨e,t⟩. We can use this same process to determine the type of V. V combines with NP_2 to produce the VP. NP_2 does not denote a function. Its type is e. Therefore, V's type must be a function that combines with the type of NP_2 to produce a meaning of the type of the VP. That is, V must denote a function that combines with an e-type expression to produce an ⟨e,t⟩-type expression. From this we can see that the type of V must be ⟨e,⟨e,t⟩⟩.



This process of determining types is similar to the process we went through earlier to determine the meaning of *sees*. It is simpler, however, since we only have to look at one property of the meaning at each step.

One thing that semantics has to do is to assign meanings to all lexical items. In many cases, the meaning of an expression is not intuitively clear. We have a clear intuition about what *cat* means. However, very few people have a clear intuition about what *the* means. To assign a meaning to the word *the*, it can be helpful to first determine its type.

The word *cat* essentially divides the world into two groups: those that are cats and those that are not. In this way, *cat* acts as a one-place predicate. If we take all one-place predicates to have the same semantic type, then *cat* is of type ⟨e,t⟩, just like *sits* and *happy*. What, then, is the type of *the* in the expression *the cat*? Notice that *the cat* can occur anywhere that a name can occur. This suggests that the semantic type of *the cat* is the same as the semantic type of a name, namely type e. If we accept these assumptions, then *the* must be a function from things of type ⟨e,t⟩ to things of type e. That is, its type must be ⟨⟨e,t⟩,e⟩:



This does not tell us yet what the full meaning of *the* is. However, it does tell us what kind of meaning *the* should have if our assumptions are correct.

The conclusion we just came to about the semantic type of *the* depends on two assumptions. One assumption was that the semantic type of *cat* is the same as the semantic type of *sits* and *happy*. This is a questionable assumption. Each of these expressions can be used to divide the individuals of the world into two groups. The expression is true of the individuals in one of the groups and false of the individuals in the other. This is the reason for assuming that all three expressions are of the same semantic type. However, there are clear differences between the three expressions as well.

If the expressions have the same semantic type, we might expect that they should all be able to occur in the same places. As seen in the following examples, however, they cannot.

Every **cat** sleeps.	Felix is very **happy**.	It seems Felix **sits**.
*Every **sits** sleeps.	*Felix is very **sits**.	*It seems Felix **cat**.
*Every **happy** sleeps.	*Felix is very **cat**.	*It seems Felix **happy**.

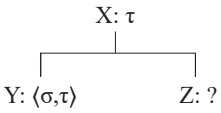
Here we have 3 sets of sentences. The top sentences are all acceptable. The results of substituting *sits* or *happy* for *cat* in the first column, though, are unacceptable, and similar substitutions are unacceptable in the second and third columns as well. Does this show that *sits*, *happy* and *cat* are of different semantic types? The answer is not clear.

We want our grammar as a whole to predict that the sentences on the top line are acceptable and that those on the second and third lines are not. There are many grammars that can do this. *Cat* is a noun, *happy* is an adjective, and *sits* is a verb. A grammar can account for the acceptability of the sentences above by restricting where nouns, adjectives and verbs can occur in the syntax. If **Every sits sleeps* breaks the rules of syntax, for example, there is no need to also explain why it is bad in the semantics. This approach would allow us to analyze nouns, adjectives and verbs as all having the same semantic type, as we have done.

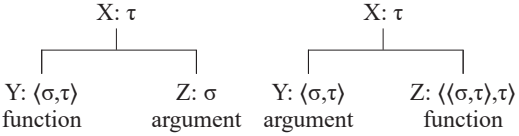
A grammar could also account for the differences semantically by assigning different semantic types to nouns, adjectives and verbs. In such a grammar, the syntax could produce all 9 of the sentences above. However, the semantics would only be able to interpret the sentences in the top row. Such a grammar would give up the assumption that nouns, adjectives and verbs can all be of type ⟨e,t⟩. There are good reasons to adopt this kind of a grammar. However, showing those reasons is a topic for a more advanced course. In this textbook we analyze all three as being of type ⟨e,t⟩.

A second assumption we made in determining the type of *the* was that the semantic type of *the cat* is type e. This is a reasonable assumption. It is not, however, a necessary assumption. Consider the sentence *The cat sleeps*. Assume that *sleeps* is of type ⟨e,t⟩ and that the sentence as a whole is of type t. Does it follow that *the cat* is of type e? No. There is another solution that works as well: *the cat* could be of type ⟨⟨e,t⟩,t⟩. If this is the semantic type of *the cat*, then *the cat* acts as a function taking *sleeps* as its argument. This reverses the function-argument relation that results if *the cat* is of type e.

Generalizing, suppose we have an expression $[_X$ Y Z] with the semantic types of X and Y as indicated below.



Then there are in principle two possible semantic types that Z could have. Z could be of type σ, or it could be of type ⟨⟨σ,τ⟩,τ⟩.



We will see in Chapter 6 that both solutions are used in semantics.

The assumptions we made in determining the semantic types of *the, cat*, and *the cat* are reasonable. Like all assumptions, however, they are possibly wrong. In doing

science, this possibility must always be kept in mind. In many cases, the details might not matter. In order to understand the word *very*, for example, it probably does not matter whether *the cat* denotes an individual (of type e) or a function (of type ⟨⟨e,t⟩,t⟩). For understanding the interpretation of the word *the*, in contrast, those same details will be all important.