# Adjectival Predicates

Lecture 08

February 6, 2024

**Announcements:**

This lecture is supplemented by the following readings:

- Heim & Kratzer: Ch.2 (34–39), Ch.4 (61–63)

Your third homework assignment is available and is due on February 8th.

## 1 Introduction

We have up to this point built a semantic system for interpreting sentences that relies on the following set of rules:

(1) **Functional Application (FA)**
If X is a node that has two daughters, Y and Z, and if $[\![ \text{Y} ]\!]$ is a function whose domain contains $[\![ \text{Z} ]\!]$, then $[\![ \text{X} ]\!] = [\![ \text{Y} ]\!]([\![ \text{Z} ]\!])$.

(2) **Non-Branching Nodes (NN) Rule**
If X is a non-branching node that has Y as its daughter, then $[\![ \text{X} ]\!] = [\![ \text{Y} ]\!]$

(3) **Terminal Nodes (TN) Rule**
If X is a terminal node, then $[\![ \text{X} ]\!]$ is specified in the lexicon.

These rules are able to interpret structures built from lexical entries like the following:

(4) **Proper Nouns as entities**
$[\![ \text{NAME} ]\!]$ = the thing referred to with NAME

(5) **Intransitive predicates as functions**
$[\![ \text{VERB}_{intrans} ]\!] = f : D_e \to D_t$
for every $x \in D_e$ }, $f(x) = T$ *iff* $x$ VERBs

(6) **Transitive predicates as function-valued functions**
$[\![ \text{VERB}_{trans} ]\!] = f : D_e \to D_{\langle e,t \rangle}$
for every $y \in D_e,\ f(y) = g : D_e \to D_t$
for every $x \in D_e,\ g(x) = T$ *iff* $x$ VERBs $y$

One of our goals for today is to introduce a standard and significantly more compact notation that is used in the field to represent functions. This **lambda notation** encodes all of the information expressed by our standard notation, but it capable of doing so much more efficiently:

(7) **Lambda notation for $\langle e, t\rangle$ predicates**

$$\begin{aligned}[\![ \text{dances} ]\!] &= [\ \lambda x : x \in D_e\ .\ T \textit{ iff } x \text{ dances}\ ] \\ &= [\ \lambda x : x \in D_e\ .\ x \text{ dances}\ ] \\ &= [\ \lambda x_e\ .\ x \text{ dances}\ ]\end{aligned}$$

(8) **Lambda notation for $\langle e, \langle e, t\rangle\rangle$ predicates**

$$\begin{aligned}[\![ \text{likes} ]\!] &= [\ \lambda y : y \in D_e\ .\ [\ \lambda x : x \in D_e\ .\ T \textit{ iff } x \text{ likes } y\ ]] \\ &= [\ \lambda y : y \in D_e\ .\ [\ \lambda x : x \in D_e\ .\ x \text{ likes } y\ ]] \\ &= [\ \lambda y_e\ .\ [\ \lambda x_e\ .\ x \text{ likes } y\ ]]\end{aligned}$$

With this hand, we will turn our attention over the next several weeks to the semantics of adjectives, adjectival modification, and more complex nominal expressions. We will start with simple cases like those below:

(9) Felix is **gray**.

(10) Felix is **a cat**.

With a focus first on adjectival predicates, we will develop a lexical entry for adjectives, like *gray*, that treat them as type $\langle e, t\rangle$ functions, just like intransitive verbs.

(11) **The semantics of adjectives**

a. **Function notation**

$[\![ \text{gray} ]\!] = f : D_e \to D_t$
for every $x \in D_e$ }, $f(x) = T$ *iff* $x$ is gray

b. **Lambda Notation**

$[\![ \text{gray} ]\!] = [\ \lambda x : x \in D_e\ .\ T \textit{ iff } x \text{ is gray}\ ]$

Interpreting sentences like those above will also require us to have a lexical entry for the copula verb *is*. We will see good reason for thinking that such elements are semantically vacuous, allowing them, and other elements, to be modeled as **identity functions**.

(12) **The copula as an identity function**

a. **Function Notation**

$[\![ \text{is} ]\!] = g : D_{\langle e,t\rangle} \to D_{\langle e,t\rangle}$
for every $f \in D_{\langle e,t\rangle}$, $g(f) = f$

b. **Lambda Notation**

$[\![ \text{is} ]\!] = [\ \lambda f : f \in D_{\langle e,t\rangle}\ .\ f]$

## 2 Lambda Notation

During our last meeting we introduced **semantic type** notation in order to develop a convenient way of categorizing and talking about the extension of expressions, to visualize semantic composition, and to reason out new semantic extensions.

This notation also gave a way to more succinctly represent the extensions of predicates:

(13) **Shorthand notation for intransitive predicates**

$[\![$ dances $]\!] = f : D_e \to D_t$

for every $x \in D_e,\ f(x) = T$ *iff* $x$ dances

(14) **Shorthand notation for transitive predicates**

$[\![$ likes $]\!] = f : D_e \to D_{\langle e,t\rangle}$

for every $x \in D_e,\ f(x) = g : D_e \to D_t$

for every $y \in D_e,\ g(y) = T$ *iff* $y$ likes $x$

Despite the simplification, these extensions can be fairly complex and cumbersome to work with, especially in the case of function-valued functions.

Therefore semanticists standardly make use of acompact notation for functions that we can refer to as **lambda notation**.

### 2.1 Introducing Lambda Notation and Lambda Conversion

**Lambda Notation.** Lambda notation simply provides yet another, but extremely compact and efficient way to represent the kinds of functions that we are already familiar with.

(15) a. **Functions as sets of ordered pairs**

$f = \{ \langle 1,4\rangle, \langle 2,5\rangle, \langle 3,6\rangle \}$

b. **Description notation of functions**

$f : \{1,2,3\} \to \{4,5,6\}$

for every $x \in \{1,2,3\},\ f(x) = x + 3$

c. **Lambda notation of functions**

$[\lambda x : x \in \{1,2,3\} \,.\, x + 3]$

Lambda notation for functions has a general schematic template that can be represented in the following way. Observe that each part of the description notation for functions finds a correlate in this new lambda notation.

(16) **Lambda Notation for Functions**

$$[\,\lambda \underbrace{x}_{\textit{argument variable}} : \underbrace{x \in D}_{\textit{domain condition}} \,.\, \underbrace{\ldots x \ldots}_{\textit{value condition}}\,]$$

"The function which takes an argument $x$, such that $x$ is in the domain $D$, and maps x to the value specified by the condition applied to $x$"

We can see these pieces in a another kind of familiar example that illustrates the use of lambda notation for functions that is provided in (17):

(17) **Illustration of lambda notation**

a. $g = \{ \langle \text{Anne, yellow} \rangle, \langle \text{Ben, orange} \rangle, \langle \text{Cynthia, red} \rangle, \dots \}$

b. $g : D_e \rightarrow \{ x : x \text{ is a color} \}$
for every $y \in D_e$, $g(y) =$ the favorite color of $y$

c. $[\lambda y : y \in D_e \text{ . the favorite color of } y]$

**Lambda Conversion.** Of course, one of the primary jobs of a function is to take an argument in order to return a value. When a function takes an argument we write the following:

(18) **Functions taking arguments in lambda notation**

$$\underbrace{[\lambda x : x \in D \,.\, \dots x \dots]}_{function} \; \underbrace{(\boldsymbol{y})}_{argument}$$

Using the examples presented above, we can see how functions are applied to their arguments in this new lambda notation:

(19) $[\lambda x : x \in \{1, 2, 3\} \,.\, x + 3](2) = 5$

(20) $[\lambda x : x \in D_e \text{ . the favorite color of } x](\text{Cynthia}) = \text{red}$

When a function is applied to an argument in **lambda notation**, we calculate the resulting value by performing a rule of **Lambda Conversion**:

(21) **Lambda Conversion (LC)**

$[\, \lambda x : x \in D \,.\, \dots x \dots ](y) = z$

(22) **Steps to Lambda Conversion**

i. *Start with the function and its argument*
$[\, \lambda x : x \in D \,.\, \dots x \dots ](y) =$

ii. *Replace all instances of the argument variable with the argument*
$[\, \lambda x : x \in D \,.\, \dots y \dots ](y)$

iii. *Remove everything but the domain condition*
~~$[\, \lambda x : x \in D \,.$~~ $\dots y \dots$ ~~$](y)$~~

iv. *Interpret the domain condition*
$\dots y \dots$

Let us consider the examples above one more time to see how Lambda Conversion works to deliver the value returned by applying the function to its argument:

(23) **Illustration of Lambda Conversion 1**

| | | | |
|---|---|---|---|
| i. | $[\lambda x : x \in \{1,2,3\} \,.\, x+3](2)$ | $=$ | (by LC) |
| ii. | $2+3$ | $=$ | (by math) |
| iii. | $5$ | | |

(24) **Illustration of Lambda Conversion 2**

| | | | |
|---|---|---|---|
| i. | $[\lambda x : x \in D_e$ . the favorite color of $x]$(Cynthia) | $=$ | (by LC) |
| ii. | the favorite color of Cynthia | $=$ | (by facts of the world) |
| iii. | red | | |

**Nested Lambda Expressions.** The real advantage of this notation comes from its ability to clearly and succinctly represent function-valued functions and other complex expressions.

To represent a function that returns a function as its value, we simply embed one lambda expression inside of another one.

(25) **Illustration of a function-valued function in lambda notation**

$[\lambda x : x \in \mathbb{N} \,.\, [\lambda y : y \in \mathbb{N} \,.\, x+y]]$

"The function which takes a number $x$ as an argument and returns
the function which takes a number $y$ as an argument and returns $x+y$"

This is a significantly more compact and readable expression than the notation that we had been working with previously. Compare the function definition below:

(26) **Illustration of a function-valued function in definition notation**

$f : \mathbb{N} \to D_{\langle \mathbb{N},\mathbb{N} \rangle}$
for every $x \in \mathbb{N},\ f(x) = g : \mathbb{N} \to \mathbb{N}$
for every $y \in \mathbb{N},\ g(y) = x + y$

Our compact lambda notation also allows us to write expressions like the following to represent how each function of the complex expression is applied to its argument:

(27) **Complex lambda expression taking arguments**

$[\lambda x : x \in \mathbb{N} \,.\, [\lambda y : y \in \mathbb{N} \,.\, x+y]](3)(4)$

The rule of Lambda Conversion operates on complex lambda expressions iteratively in exactly the same way as we saw above. Note that the convention is to list arguments in the order in which they are applied.

(28) **Interpreting complex lambda expressions**

| | | | |
|---|---|---|---|
| i. | $[\lambda x : x \in \mathbb{N} . [\lambda y : y \in \mathbb{N} . x + y]](3)(4)$ | = | (by **LC**) |
| ii. | $[\lambda y : y \in \mathbb{N} . 3 + y](4)$ | = | (by **LC**) |
| iii. | $3 + 4$ | = | (by math) |
| iv. | $7$ | | |

## 2.2 Lambda Notation in Truth-Conditional Semantics

Lambda notation for expressing functions can be adapted relatively straightforwardly to represent meanings in our semantic system.

Recall that the extension of a predicate is its the truth value determined by the conditions specified by the predicate.

(29) **Shorthand function notation for predicates**

$[\![\text{ dances }]\!] = f : D_e \to D_t$
for every $x \in D_e, f(x) = T$ *iff* $x$ dances

**Lambda Notation for Predicates.** These pieces of our standard notation are translated into lambda notation just as shown below:

(30) **Type $\langle e, t \rangle$ functions in lambda notation**

$$[\, \lambda \underbrace{x}_{\textit{argument variable}} : \underbrace{x \in D_e}_{\textit{domain condition}} . \underbrace{T \textit{ iff } x \text{ dances}}_{\textit{value condition}} \,]$$

Even as compact as such expressions are, we will occasionally find that it is useful or convenient to abbreviate these representations in the following ways:

(31) **Some notational equivalencies for the lambda expression of predicates**

a. $[\![\text{ dances }]\!] = [\lambda x : x \in D_e . T$ *iff* $x$ dances $]$
b. $= [\lambda x : x \in D_e . x$ dances $]$ (“$T$ *iff*” omitted from value condition)
c. $= [\lambda x \in \boldsymbol{D_e} . x$ dances $]$ (domain condition added to argument variable)
d. $= [\lambda x_{\boldsymbol{e}} . x$ dances $]$ (domain condition as subscript on argument variable)
e. $= [\lambda x . x$ dances $]$ (domain condition ‘too obvious’ to include)

Using nested lambda expression in the same way as we did above, it is possible to move on from the cumbersome expressions like those below:

(32) **Shorthand notation for transitive predicates**

$[\![\text{ likes }]\!] = f : D_e \to D_{\langle e,t \rangle}$
for every $x \in D_e,\ f(x) = g : D_e \to D_t$
for every $y \in D_e,\ g(y) = T$ *iff* $y$ likes $x$

Instead, we can represent the function-valued functions of the type that we discovered for transitive predicates by embedding on function inside of another:

(33) **Type $\langle e, \langle e, t \rangle\rangle$ functions in lambda notation**

$$[\lambda \underbrace{x}_{\textit{argument variable}} : \underbrace{x \in D_e}_{\textit{domain condition}} . \underbrace{[\lambda \underbrace{y}_{\textit{argument variable}} : \underbrace{y \in D_e}_{\textit{domain condition}} . \underbrace{T \textit{ iff } y \text{ likes } x}_{\textit{value condition}}]}_{\textit{value condition}}]$$

The same space-saving conventions that we used above can be used here as well:

(34) **Some notational equivalences for the lambda expression of transitive verbs**

a. $[\![ \text{likes} ]\!] = [\lambda x : x \in D_e . [\lambda y : y \in D_e . T \textit{ iff } y \text{ likes } x]]$

b. $\quad = [\lambda x \in D_e . [\lambda y \in D_e . y \text{ likes } x]]$

c. $\quad = [\lambda x_e . [\lambda y_e . y \text{ likes } x]]$

**Predicates Taking Arguments.** We are able to indicate functions taking argument using the same notation that we adopted above:

(35) **Lambda notation for predicates taking arguments**

$$[\![ \text{dances} ]\!]([\![ \text{Sam} ]\!]) = \underbrace{[\lambda x : x \in D_e . T \textit{ iff } x \text{ dances} ]}_{function} \underbrace{(\text{Sam})}_{argument}$$

The rule of **Lambda Conversion** allows us to calculate the value the results from applying a predicate to its argument:

(36) **Illustration of Lambda Conversion with predicates**

| | | | |
|---|---|---|---|
| i. | $[\![ \text{dances} ]\!]([\![ \text{Sam} ]\!])$ | = | (by TN x 2) |
| ii. | $[\lambda x : x \in D_e . T \textit{ iff } x \text{ dances} ](\text{Sam})$ | = | (by **LC**) |
| iii. | $T$ *iff* Sam dances | | |

(37) **Illustration of Lambda Conversion with predicates**

| | | | |
|---|---|---|---|
| i. | $[\![ \text{dances} ]\!]([\![ \text{Sam} ]\!])$ | = | (by TN x 2) |
| ii. | $[\lambda x_e . x \text{ dances} ](\text{Sam})$ | = | (by **LC**) |
| iii. | $T$ *iff* Sam dances | | |

It is important to keep in mind that the extension of functions that return truth values is the the **truth value**. Thus, when Lambda Conversion is applied to such expressions, the result will always be truth-conditional statements like in (36) and (37).

This means that we will never perform Lambda Conversion to generate things like the following:

(38) **Incorrect usage of Lambda Conversion with predicates**

i. $[\lambda x : x \in D_e . T \textit{ iff } x \text{ dances }](\text{Sam}) \quad \neq \quad$ (by LC)

ii. Sam dances

(39) **Incorrect usage of Lambda Conversion with predicates**

i. $[\lambda x : x \in D_e . x \text{ dances }](\text{Sam}) \quad \neq \quad$ (by LC)

ii. Sam dances

Finally, we can demonstrate how function-valued functions combine with a sequence of arguments.

(40) **Illustration of Lambda Conversion with transitive predicates**

i. $[\![ \text{ likes } ]\!]([\![ \text{ Taylor } ]\!])([\![ \text{ Travis } ]\!]) \quad = \quad$ (by TN x 3)

ii. $[\lambda x : x \in D_e . [\lambda y : y \in D_e . T \textit{ iff } y \text{ likes } x]](\text{Taylor})(\text{Travis}) \quad = \quad$ (by $\underline{\textbf{LC}}$)

ii. $[\lambda y : y \in D_e . T \textit{ iff } y \text{ likes Taylor }]](\text{Travis}) \quad = \quad$ (by $\underline{\textbf{LC}}$)

iii. $T$ *iff* Travis likes Taylor

(41) **Illustration of Lambda Conversion with transitive predicates**

i. $[\![ \text{ likes } ]\!]([\![ \text{ Taylor } ]\!])([\![ \text{ Travis } ]\!]) \quad = \quad$ (by TN x 3)

ii. $[\lambda x_e . [\lambda y_e . y \text{ likes } x]](\text{Taylor})(\text{Travis}) \quad = \quad$ (by $\underline{\textbf{LC}}$)

ii. $[\lambda y_e . y \text{ likes Taylor }]](\text{Travis}) \quad = \quad$ (by $\underline{\textbf{LC}}$)

iii. $T$ *iff* Travis likes Taylor

Moving forward, we will employ the lambda notation for functions in our semantic system for computing the meaning of expressions.

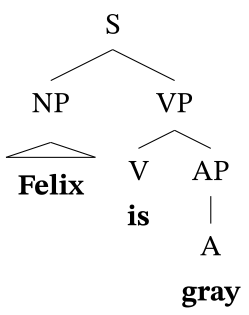
## 3 The Semantics of the Copula

Let's now return to our project of further developing our semantic system to compute the meaning of sentences with adjectival predicates.

(42) Felix is gray.

Regarding the syntax of such constructions, let's assume that (42) has a syntactic representation. The copula will be treated as a verb that combines with an AP headed by *gray*.

(43)



In order for our system to interpret this sentence, we will need a lexical entry for both the copula *is* and the adjective *gray*. Furthermore, using these lexical entries, we will need to be able to compute the meaning of the VP and AP nodes.

We will start by determining a lexical entry for the copula.

## 3.1 The Semantic Type of the Copula

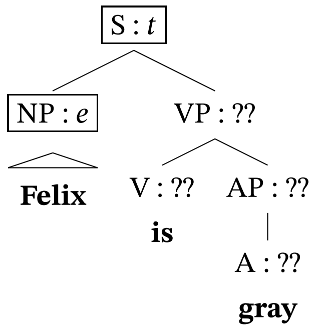
**1. Known Semantic Types.** We start by listing out the semantic types we know and annotating the syntactic representation with this information.

(44) **Known Semantic Types in (43)**

a. $[\![\text{S}]\!] \in D_t$

b. $[\![\text{Felix}]\!] \in D_e$

(45)



**2. Reasoning out the Semantic Type of VP.** We can appeal to these known semantic types, as well as the rules of composition, to determine that the VP must be of type $\langle e, t\rangle$.

- The sentence must ultimately be an expression of type $t$.
- The subject NP *Felix* is an expression of type $e$.
- The S node fits the description for Functional Application. So, the extension of the subject must combine with the extension of the VP to return the extension of the sentence.

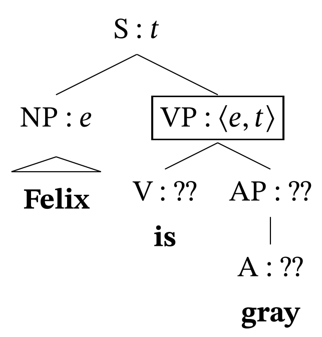
  (46) $[\![\text{Felix}]\!] + [\![\text{VP}]\!] = [\![\text{S}]\!]$

- Because the NP *Felix* is an expression of type $e$, the extension of the VP must be a function that takes *Felix* as an argument and gives back a truth value.

  (47) $f : D_e \rightarrow D_t$

- Thus, **the VP must be an expression of type $\langle e, t\rangle$**, which happens to be a type of expression that we are already familiar with.

(48)



**3. Reasoning out the Semantic Type of the Copula.** Here's where we run into a problem.

- Given our syntactic assumptions above and our current rules of composition (*viz.*, FA), the extension of *is* and the extension of the AP must combine to give us the extension of the VP.

  (49) $[\![ \text{is} ]\!] + [\![ \text{AP} ]\!] = [\![ \text{VP} ]\!]$

- Neither the copula nor the AP is plausibly treated as an expression of type *e*.
- It is not clear how the copula and the AP compose to produce the extension of the VP.

## 3.2 Introducing Identity Functions

An influential idea for this problem is to say that the copula verb is in fact **semantically vacuous**. That is, it appears in the sentence for purely syntactic reasons, like expressing tense, and does not contribute to the meaning of the expression.

Traditional grammarians have long held this view of copulas. We were probably taught at some point that the copula *be* is a "helping" verb that links subjects and adjectives. In fact, the term *copula* has its roots in Latin with the meaning "joiner" or "coupler".

Moreover, there are many languages, including English, that do not require a copula or lack it altogether.

(50) **Russian Null-Copula Constructions**

a. Felix s$^{j}$irɨj
   Felix gray.MS
   'Felix is gray.'

b. Felix kot
   Felix cat
   'Felix is a cat.'

This is consistent with the claim that the copula doesn't really add anything to the meaning of an English sentence. It is there purely for "surface" reasons.

But even if the copula "doesn't add anything to the meaning of the sentence," it must still participate in the composition of the sentence. This leads us to the concept of so-called **identity functions**.

(51) **Identity Function**
A function that takes an argument *x* and returns *x* as its value.

With identity functions in our semantics toolbox, let's try again to reason out a semantic type for the lexical entry of the copula.

**Reasoning out the Semantic Type of the Copula: Second Try.** We can appeal to our known semantic types, as well as the rules of composition, to reason out a semantic type for the copula that makes it an identity function.

- The VP node was just determined in section 3.1 to be an expression of type $\langle e, t\rangle$.
- The copula is to be treated as an identity function that returns a value with the same semantic type as its argument.
- The VP node fits the structural description for Functional Application. So, the extension of the copula must combine with the extension of the AP to return the extension of the VP.

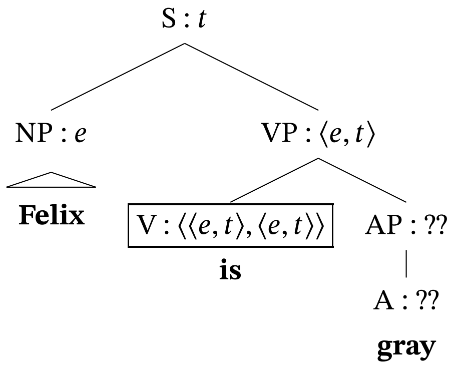
(52) $[\![\text{ is }]\!] + [\![\text{ AP }]\!] = [\![\text{ VP }]\!]$

- Because the VP is an expression of $\langle e, t\rangle$, the extension of **the copula must be a function that takes a $\langle \boldsymbol{e}, \boldsymbol{t}\rangle$ function as its argument and returns the same $\langle \boldsymbol{e}, \boldsymbol{t}\rangle$ function**.

(53) $f : D_{\langle e,t\rangle} \rightarrow D_{\langle e,t\rangle}$

- Thus, the copula must be an identity function of type $\langle\langle e, t\rangle, \langle e, t\rangle\rangle$.

(54)


S : $t$
NP : $e$
VP : $\langle e, t\rangle$
Felix
V : $\langle\langle e, t\rangle, \langle e, t\rangle\rangle$
is
AP : ??
A : ??
gray

## 3.3 The Extension of the Copula

With this much in hand, it is relatively straightforward to propose the extension of the copula verb.

As a type $\langle\langle e, t\rangle, \langle e, t\rangle\rangle$ identify function, the lexical entry of the copula *is* can be represented, in both our old and new notations, as shown in (55).

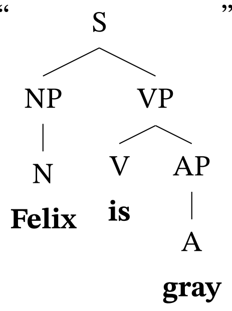
(55) **The copula as an identity function**

a. **Function Notation**
$[\![\text{ is }]\!] = g : D_{\langle e,t\rangle} \rightarrow D_{\langle e,t\rangle}$
for every $f \in D_{\langle e,t\rangle}$, $g(f) = f$

b. **Lambda Notation**
$[\![\text{ is }]\!] = [\, \lambda f : f \in D_{\langle e,t\rangle} \,.\, f]$
"The function which takes a type $\langle e, t\rangle$ function as an argument and returns the same type $\langle e, t\rangle$ function."

## 4 The Semantics of Adjectives

Having decided on our syntactic representation for the sentence at hand and having determined an extension for the copula, we can proceed with our goal of deriving the following truth-conditional statement:

(56) “ 

” is *T iff* Felix is gray

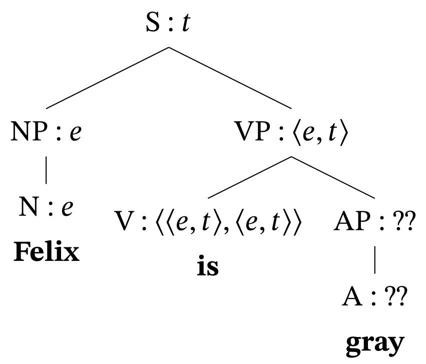
This will require that we determine the extension of the adjective *gray* and demonstrate that our semantic system is capable of deriving these truth conditions.

### 4.1 The Semantic Type of Adjectives

**1. Known Semantic Types.** We start by listing out the semantic types we know and annotating the syntactic representation with this information.

(57) **Known Semantic Types**

| | | | |
|---|---|---|---|
| a. | $[\![ \text{S} ]\!]$ | $\in$ | $D_t$ |
| b. | $[\![ \text{Felix} ]\!]$ | $\in$ | $D_e$ |
| c. | $[\![ \text{is} ]\!]$ | $\in$ | $D_{\langle\langle e,t\rangle,\langle e,t\rangle\rangle}$ |

(58) 

**2. Reasoning out the Semantic Type of AP.** We appeal to our known semantic types, as well as the rules of composition, to determine that the AP must be of type $\langle e, t\rangle$.

- The VP node fits the structural description for Functional Application. So, the extension of the copula must combine with the extension of the AP to return the extension of the VP.

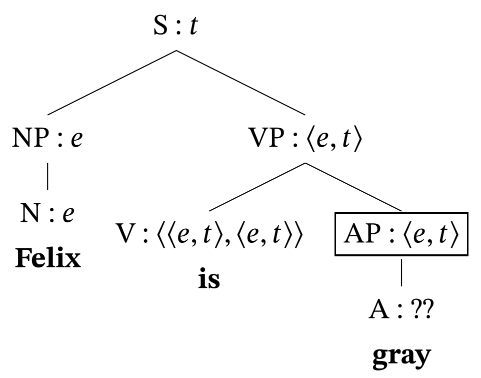
  (59) $[\![ \text{is} ]\!] + [\![ \text{AP} ]\!] = [\![ \text{VP} ]\!]$

- Because the copula is a function of type $\langle\langle e,t\rangle,\langle e,t\rangle\rangle$, the extension of the copula must take the AP as its argument to return the type $\langle e, t\rangle$ extension of the VP as its value.

  (60) $[\![ \text{is} ]\!]([\![ \text{AP} ]\!]) = [\![ \text{VP} ]\!]$

- Thus, **the AP must be an expression of type $\langle e, t\rangle$.**

(61)



S : $t$
NP : $e$
VP : $\langle e, t\rangle$
N : $e$
Felix
V : $\langle\langle e, t\rangle, \langle e, t\rangle\rangle$
is
$\boxed{\text{AP} : \langle e, t\rangle}$
A : ??
gray

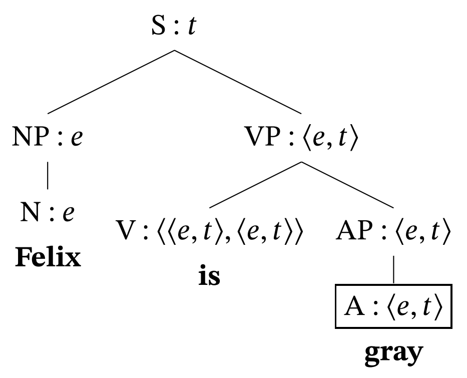
**3. Reasoning out the Semantic Type of A.** We appeal to our known semantic types, as well as the rules of composition, to determine that the A must be of type $\langle e, t\rangle$.

- We just determined that the AP node must be an expression of type $\langle e, t\rangle$.
- The AP node has the structural description specified by the Non-Branching Node Rule. So, the extension of the AP must be equivalent to the extension of the A.

(62) $[\![ \text{AP} ]\!] = [\![ \text{A} ]\!]$

- Thus, **the A must be an expression of type $\langle e, t\rangle$.**

(63)



S : $t$
NP : $e$
VP : $\langle e, t\rangle$
N : $e$
Felix
V : $\langle\langle e, t\rangle, \langle e, t\rangle\rangle$
is
AP : $\langle e, t\rangle$
$\boxed{\text{A} : \langle e, t\rangle}$
gray

## 4.2 The Extension of Adjectives

We now know what kind of expression the A *gray* must be. It is a type $\langle e, t\rangle$ function, meaning it maps entities to truth values.

We therefore need to develop a lexical entry for *gray* that does the following:

- assigns as its extension a function of type $\langle e, t\rangle$ and
- allows our system to derive the following truth-conditional statement:

(64) "Felix is gray" is $T$ *iff* Felix is gray

**Some Preliminary Reasoning about the Extension of A.** On the basis of what has preceded, we can appreciate that the extension of the VP will be equivalent to the extension of the A.

- We know from section 3 that the type $\langle e,t\rangle$ extension of the VP will be equivalent to the type $\langle e,t\rangle$ extension of the AP on account of the copula's extension as an identity function.
- We know from section 4.1 that the type $\langle e,t\rangle$ extension of the AP will be equivalent to the type $\langle e,t\rangle$ extension of the A on account of the Non-Branching Nodes Rule.
- From these two points it follows that the extension of the VP will be equivalent to the extension of the A.

(65) **The extension of the VP is identical to the extension of the A**

| | | | |
|---|---|---|---|
| i. | $[\![\ [_{\text{VP}} \text{ is gray }]\ ]\!]$ | = | (by FA) |
| ii. | $[\![\text{ is }]\!]([\![\text{ AP }]\!])$ | = | (by NN) |
| ii. | $[\![\text{ is }]\!]([\![\text{ gray }]\!])$ | = | (by TN) |
| iii. | $[\,\lambda f : f \in D_{\langle e,t\rangle} \,.\, f\,]([\![\text{ gray }]\!])$ | = | (by LC) |
| iv. | $[\![\text{ gray }]\!]$ | | |

**Reasoning out the Extension of the A.** Given the results above, determining the meaning of the type $\langle e,t\rangle$ VP *is gray* will deliver the meaning of the A *gray*.

- Considering sentences like those in (66), the key generalization is that the VP *is gray* systematically combines with entities to generate sentences with the meaning in (67).

(66) a. "Felix is gray" is $T$ *iff* Felix is gray
b. "Mittens is gray" is $T$ *iff* Mittens is gray
c. "Luna is gray" is $T$ *iff* Luna is gray

(67) $[\![\ [_{\text{S}} \text{ NAME is gray }]\ ]\!] = T$ *iff* NAME is gray

- We know from section 3.1 that $[\![\ [_{\text{VP}} \text{ is gray }]\ ]\!]$ is of type $\langle e,t\rangle$. Consequently, our rule of FA entails that the following equivalency holds:

(68) $[\![\ [_{\text{S}} \text{ NAME is gray }]\ ]\!] = [\![\ [_{\text{VP}} \text{ is gray }]\ ]\!]([\![\text{ NAME }]\!])$

- If follows from the previous two points that:

(69) $[\![\ [_{\text{VP}} \text{ is male }]\ ]\!]([\![\text{ NAME }]\!]) = T$ *iff* NAME is gray

- Therefore, $[\![\ [_{\text{VP}} \text{ is gray }]\ ]\!]$ is an $\langle e,t\rangle$ function that takes an argument $x$ yields $T$ *iff* $x$ is gray.

(70) $[\![\ [_{\text{VP}} \text{ is gray }]\ ]\!] = [\lambda x : x \in D_e \,.\, T \textit{ iff } x \text{ is gray }]$

- Because $[\![\ [_{\text{VP}} \text{ is gray }]\ ]\!] = [\![\ [_{\text{A}} \text{ gray }]\ ]\!]$, then:

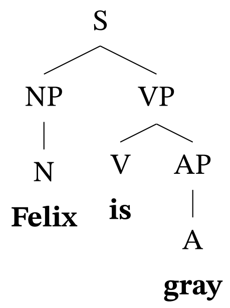
(71) $[\![\text{ gray }]\!] = [\lambda x : x \in D_e \,.\, T \textit{ iff } x \text{ is gray }]$

## 5 Deriving the Meaning of Sentences with Predicate Adjectives

Let us now check to be sure that our semantic system correctly makes use of these lexical entires to the derive the correct truth-conditional statement.

Our syntactic assumptions regarding the sentence at hand are as follows:

(72) a. Felix is gray.
b.



The lexical entries for the components parts of the sentence that are stored in the lexicon include:

(73) **Lexical entries**

a. ⟦ Felix ⟧ = Felix

b. ⟦ is ⟧ = $[\, \lambda f : f \in D_{\langle e,t \rangle} \,.\, f ]$

c. ⟦ gray ⟧ = $[\lambda x : x \in D_e \,.\, T$ *iff* $x$ is gray ]

Our system currently employs the set of semantic rules listed below:

(74) **Functional Application (FA)**
If X is a node that has two daughters, Y and Z, and if ⟦ Y ⟧ is a function whose domain contains ⟦ Z ⟧, then ⟦ X ⟧ = ⟦ Y ⟧(⟦ Z ⟧).

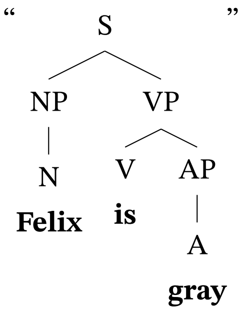
(75) **Non-Branching Nodes (NN) Rule**
If X is a non-branching node that has Y as its daughter, then ⟦ X ⟧ = ⟦ Y ⟧

(76) **Terminal Nodes (TN) Rule**
If X is a terminal node, then ⟦ X ⟧ is specified in the lexicon.

We can now see how the proof of the truth conditions can proceed with subproofs of the major constituents of the sentence.

(77) **Truth-conditional Statement of *Felix is gray***
*Felix is gray* is $T$ *iff* Felix is gray

(78) **Calculation of the Truth Conditions of *Felix is gray***

| | | | |
|---|---|---|---|
| i. | "Felix is gray" | is *T iff* | (by syntax) |
| ii. | "  " | is *T iff* | (by notation) |
| iii. | ⟦ S ⟧ | $= T$ | |
| iv. | *Calculation of* ⟦ NP ⟧ | | |
| | a. ⟦ NP ⟧ | = | (by NN) |
| | b. ⟦ Felix ⟧ | = | (by TN) |
| | c. Felix | | |
| v. | *Calculation of* ⟦ AP ⟧ | | |
| | a. ⟦ AP ⟧ | = | (by NN) |
| | b. ⟦ gray ⟧ | = | (by TN) |
| | c. $[\lambda x : x \in D_e \,.\, T$ *iff* $x$ is gray ] | | |
| vi. | *Calculation of* ⟦ VP ⟧ | | |
| | a. ⟦ VP ⟧ | = | (by FA, v.) |
| | b. ⟦ is ⟧(⟦ AP ⟧) | = | (by TN) |
| | c. $[\, \lambda f : f \in D_{\langle e,t\rangle} \,.\, f\,]$(⟦ AP ⟧) | = | (by v.) |
| | d. $[\, \lambda f : f \in D_{\langle e,t\rangle} \,.\, f\,]([\lambda x : x \in D_e \,.\, T$ *iff* $x$ is gray ]) | = | (by LC) |
| | e. $[\lambda x : x \in D_e \,.\, T$ *iff* $x$ is gray ] | | |
| vii. | ⟦ S ⟧ | $= T$ *iff* | (by FA, iv., vi.) |
| viii. | ⟦ VP ⟧(⟦ NP ⟧) | $= T$ *iff* | (by vi.) |
| ix. | $[\lambda x : x \in D_e \,.\, T$ *iff* $x$ is gray ](⟦ NP ⟧) | $= T$ *iff* | (by iv.) |
| x. | $[\lambda x : x \in D_e \,.\, T$ *iff* $x$ is gray ](Felix) | $= T$ *iff* | (by LC) |
| xi. | Felix is gray | | (nailed it) |

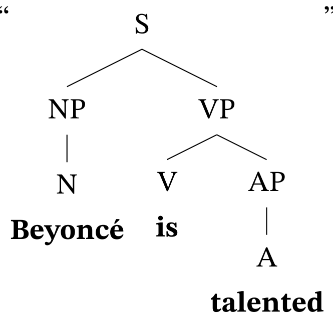
Thus, we conclude that (78i) *iff* (78xi), or **"Felix is gray" is *T iff* Felix is gray**.

By repeating the process made explicit in this handout, we would find that a similar lexical entry would work for a wide range of adjectives in English and beyond. For example, this system will similarly derive the following truth-conditional statements for several adjectives, which seem to be correct:

(79) a. "Felix is **furry**" is *T iff* Felix is **furry**.
b. "Felix is **male**" is *T iff* Felix is **male**.
c. "Felix is **single**" is *T iff* Felix is **single**.

## 6 Practice

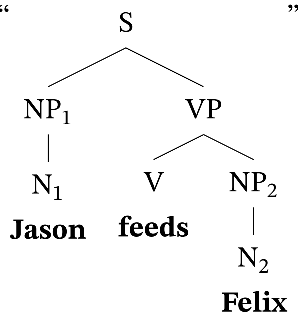
**Exercise.** Please provide a semantic proof of the truth-conditional statement below:

(80) "


" is *T iff* Beyoncé is talented

In order to do this, we will need to provide the extensions of the lexical items and provide a step-by-step proof of these reported truth conditions.

**Exercise.** Please provide a semantic proof of the truth-conditional statement below:

(81) "


" is *T iff* Jason feeds Felix

In order to do this, we will need to provide the extensions of the lexical items and provide a step-by-step proof of these reported truth conditions.