Expanding Our Formalism, Part 2

Goals for This Section:

- Introduce the ‘lambda notation’ for functions
- Practice using lambdas in derivations of T-conditions
- Introduce functions that take other functions as arguments
- Discuss the relation between syntactic ‘well-formedness’ and semantic ‘interpretability’.

1. Lambda Notation for Defining Functions

As you may have guessed by this point, most expressions of natural language will have some kind of function as their extension…

And, we’ve already seen that some of these functions can get rather complex when we try to define them in our current notation.

(1) Example: The Extension of “Or”

\[
[[ \text{or} ]] = \quad q : D_t \to D_{<1,1} \\
\text{for every } x \in D_t, \quad q(x) = p_x : D_t \to D_t \\
\text{for every } y \in D_t, p_x(y) = T \text{ iff } y=T \text{ or } x=T
\]

Isn’t there a simpler notation for defining functions???

(2) Lambda Notation, Part 1

a. Syntax: \[ \lambda x : x \in D . \varphi(x) \]

b. Semantics: The function whose domain is D (i.e., which takes as argument anything in the set D), and for all x \in D, maps x to \varphi(x)

---

1 These notes are based on the material in Heim & Kratzer (1998: 34-49).
(3) **Examples**

a. \[ \lambda x : x \in \{ 0, 1, 2, 3 \} . x + 3 \]  
   
   (i)  \{ <0,3>, <1,4>, <2,5>, <3,6> \}  
   
   (ii) f : \{ 0, 1, 2, 3 \} \rightarrow \{ 3, 4, 5, 6 \}  
        for all x \in \{ 0, 1, 2, 3 \}, f(x) = x + 3

b. \[ \lambda x : x \in \{ \text{Beatles, Rush} \} . \text{the drummer for } x \]  
   
   (i)  \{ <\text{Beatles, Ringo Starr}>, <\text{Rush, Neal Peart}> \}  
   
   (ii) g : \{ \text{Beatles, Rush} \} \rightarrow \{ y : y \text{ is a drummer} \}  
        for all x \in \{ \text{Beatles, Rush} \}, g(x) = \text{the drummer for } x

(4) **Lambda Notation: Functions Taking Arguments**

\[ \lambda x : x \in D . \varphi(x) \] (a)  

= the unique y such that <a,y> \in [ \lambda x : x \in D . \varphi(x) ]  

= the function \([ \lambda x : x \in D . \varphi(x) ]\)' taking a as argument

(5) **Examples**

a. \[ \lambda x : x \in \{ 0, 1, 2, 3 \} . x + 3 \]  
   
   (2)  
   
   = 5

b. \[ \lambda x : x \in \{ \text{Beatles, Rush} \} . \text{the drummer for } x \]  
   
   (Rush) = \text{the drummer for } \text{Rush} = \text{Neal Peart}

(6) **The Rule of ‘Lambda Conversion’ (LC)**

The following equation is a consequence of how our notation is defined…
Since we’ll be using it quite a bit, it’s nice to have a name for it: ‘Lambda Conversion’

\[ \lambda x : x \in D . \varphi(x) \](a)  

= \varphi(a)

(7) **Examples**

a. \[ \lambda x : x \in \{ 0, 1, 2, 3 \} . x + 3 \]  
   
   (2)  
   
   = 2 + 3  
   
   = 5

b. \[ \lambda x : x \in \{ \text{Beatles, Rush} \} . \text{the drummer for } x \]  
   
   (Rush) = \text{the drummer for } \text{Rush} = \text{Neal Peart}
The True Power of This Notation

- The real advantage of lambda notation is that it offers a very handy and simple way of defining functions that yield other functions as values.

- The way to represent such functions is incredibly simple: *You just embed one lambda formula inside another one!*

Example

\[
\lambda x : x \in \mathbb{N} . \left( \lambda z : z \in \mathbb{N} . x + z \right)
\]

This is the function which takes a number \( x \) as argument and returns the function which takes a number \( z \) as argument and returns \( x + z \).

Example

\[
\lambda x : x \in \mathbb{N} . \left( \lambda z : z \in \mathbb{N} . x - z \right)
\]

(by LC)

Equation That Follows From (11b)

\[
\lambda x : x \in \mathbb{N} . \left( \lambda z : z \in \mathbb{N} . x - z \right) = \lambda z : z \in \mathbb{N} . 3 - z
\]

Convention for Sequences of Arguments

Now that we can embed ‘lambdas inside of lambdas’, we can also write out formulae that look like the following:

\[
\lambda x : x \in \mathbb{N} . \left( \lambda z : z \in \mathbb{N} . x - z \right)
\]

b. How You Read The Formula Above:

- The function ‘\( \lambda x : x \in \mathbb{N} . \left( \lambda z : z \in \mathbb{N} . x - z \right) \)’ taking \( 4 \) as argument.

c. Equation That Follows From (11b)

\[
\lambda x : x \in \mathbb{N} . \left( \lambda z : z \in \mathbb{N} . x - z \right) = \lambda z : z \in \mathbb{N} . 3 - z
\]

---

\[2\] To save space, I will write ‘\( \mathbb{N} \)’ for the set \{ \( x \) : \( x \) is a whole number greater than 0 \}
(12) **Simplification of Lambda Expressions: Examples**

a.  
(i) \[
\lambda x : x \in \mathbb{N} . \left[ \lambda z : z \in \mathbb{N} . x - z \right]
\] (3)(4) = (by LC)  
(ii) \[
\lambda z : z \in \mathbb{N} . 3 - z \]
(4) = (by LC)  
(iii) 3 - 4  
(iv) -1  

b.  
(i) \[
\lambda x : x \in \mathbb{N} . \left[ \lambda y : y \in \mathbb{N} . \left[ \lambda z : z \in \mathbb{N} . (x+y) - z \right] \right]
\] (1)(5)(6)  
(ii) \[
\lambda y : y \in \mathbb{N} . \left[ \lambda z : z \in \mathbb{N} . (1+y) - z \right]
\] (5)(6)  
(iii) \[
\lambda z : z \in \mathbb{N} . (1+5) - z \]
(6)  
(iv) (1+5) - 6  
(v) 6 - 6  
(vi) 0

(13) **Crucial Question**

How do we use lambda notation to represent a function like the following?

\[ f : D_e \rightarrow D_t \]
for all \( x \in D_e \), \( f(x) = T \) iff \( x \) smokes

**Answer:** It involves a new, special part of the lambda notation.  

(14) **Lambda Notation, Part 2**

a. **Syntax:** \[
\lambda x : x \in D . \text{IF } \varphi(x) \text{ THEN } y, \text{ ELSE } z
\]

b. **Semantics:** The function whose domain is \( D \) (i.e., which takes as argument anything in the set \( D \)), and for all \( x \in D \), maps \( x \) to \( y \) if \( \varphi(x) \), and maps \( x \) to \( z \) otherwise

(15) **Example**

\[
\lambda x : x \in D_e . \text{IF } x \text{ smokes THEN } T, \text{ ELSE } F
\]  

a. The function whose domain is \( D_e \), and for all \( x \in D_e \), maps \( x \) to \( T \) iff \( x \) smokes  

b. \[[ \text{smokes} ]\]

---

3 The notation we will use for representing functions like \[[\text{smokes}]\] will (for a little while) differ from what you find in Heim & Kratzer (1998). We will eventually move to the system found in Heim & Kratzer (1998), and point out how it relates to the system we’re using here.

4 Again, this notation is not found in Heim & Kratzer (1998). A highly technical discussion of it can be found at the following: http://en.wikipedia.org/wiki/Lambda_calculus#Logic_and_predicates
(16) **Another Example**

\[ \lambda x : x \in D_e . \ \text{IF } x \ \text{dances THEN } T, \ \text{ELSE } F \] =

a. The function whose domain is \( D_e \), and for all \( x \in D_e \), maps \( x \) to \( T \) iff \( x \) dances

b. \([ [ \text{dances} ] ]\)

(17) **Still Another Example**

\[ \lambda x : x \in D_t . \ \text{IF } x = F \ \text{THEN } T, \ \text{ELSE } F \] =

a. The function whose domain is \( D_t \), and for all \( x \in D_t \), maps \( x \) to \( T \) iff \( x = F \)

b. \([ [ \text{it is not the case that} ] ]\)

(18) **Still One More Example**

\[ \lambda x : x \in D_e . \ \text{IF } x \ \text{likes Joe THEN } T, \ \text{ELSE } F \] =

a. The function whose domain is \( D_e \), and for all \( x \in D_e \), maps \( x \) to \( T \) iff \( x \) likes Joe

b. \([ [ \text{likes Joe} ] ]\)

**Key Observation:** Given that \([ [ \text{likes Joe} ] ]\) is the formula in (18), we now have at our disposal the means for representing \([ [ \text{likes} ] ]\) in our lambda notation:

(19) **The Key Example**

\[ \lambda y : y \in D_e . \ [ \lambda x : x \in D_e . \ \text{IF } x \ \text{likes y THEN } T, \ \text{ELSE } F ] \] =

a. The function whose domain is \( D_e \) and for all \( y \in D_e \), maps \( y \) to…

   the function whose domain is \( D_e \), and for all \( x \in D_e \), maps \( x \) to \( T \) iff \( x \) likes \( y \)

b. \([ [ \text{likes} ] ]\)
(20) Sample Derivation of Truth-Conditions Using Lambdas

a. "S" is T iff (by notation)

```
        NP₁
         ∣
         N₁  V  NP₂
         ∣    ∣   ∣
      likes  N₂   Joe
Barack
```

b. \[ [S] \] = T

c. Subproof

(i) \[ [[NP₁]] = \] (by NN)

(ii) \[ [[N₁]] = \] (by NN)

(iii) \[ [[Barack]] = \] (by TN)

(iv) Barack

d. Subproof

(i) \[ [[NP₂]] = \] (by NN)

(ii) \[ [[N₂]] = \] (by NN)

(iii) \[ [[Joe]] = \] (by TN)

(iv) Joe

e. Subproof

(i) \[ [[V]] = \] (by NN)

(ii) \[ [[\text{likes}]] = \] (by TN)

(iii) \[ \lambda y : y \in D_e . [ \lambda x : x \in D_e . \text{IF} x \text{ likes } y \text{ THEN } T, \text{ ELSE } F ] \]

f. Subproof

(i) \[ [[\text{VP}]] = \] (by FA, d, e)
(ii) \([V][[NP_2]] = \) (by d)

(iii) \([V](\text{Joe}) = \) (by e)

(iv) \([\lambda y : y \in D_e \cdot [\lambda x : x \in D_e \cdot \text{IF } x \text{ likes } y \text{ THEN } T, \text{ ELSE } F ]](\text{Joe}) = \) (by LC)

(v) \([\lambda x : x \in D_e \cdot \text{IF } x \text{ likes Joe THEN } T, \text{ ELSE } F ]\)

g. \([S] = T \text{ iff} \) (by FA, c, f)

h. \([VP][[NP_1]] = T \text{ iff} \) (by c)

i. \([VP](\text{Barack}) = T \text{ iff} \) (by f)

j. \([\lambda x : x \in D_e \cdot \text{IF } x \text{ likes Joe THEN } T, \text{ ELSE } F ](\text{Barack}) = T \text{ iff} \) (by LC)

k. Barack likes Joe.

(21) **Some Important Abbreviations**

Compact as our lambda notation is, we’ll occasionally want to use even shorter formulae when certain information is already clear from context.

\([\lambda y : y \in D_X \cdot \ldots ] = \) (i) \([\lambda y \in D_X : \ldots ]\)

(ii) \([\lambda y x : \ldots ]\)

(iii) *when it’s clear from context what the domain of the function is, we can even just write:

\([\lambda y : \ldots ]\)*

(22) **Example**

\([\lambda y_t : [\lambda x_t : \text{IF } x = T \text{ or } y = T \text{ THEN } T, \text{ ELSE } F ]] = \)

a. The function whose domain is D_t and for all y \in D_t, maps y to…
   the function whose domain is D_t, and for all x \in D_t, maps x to T iff x=T or y=T

b. \([\text{ or }]\)
2. **Functions That Take Other Functions as Arguments**

Our lambda notation allows us construct a wide range of functions to serve as the extensions of natural language expressions.

So far, the functions we’ve constructed have taken either entities or T-values as arguments...

...but we can also construct functions that take *other functions* as arguments!

(23) **A Simple (but Artificial) Example**

\[ \lambda f : f \in D_{<et>}. f(Barack) \] = The function from <et> functions to T-values, which for any function f, yields the value f(Barack)

(24) **A More Complex, but Realistic Example**

The connective “and” in English can conjoin two VPs. *What is its semantics in such a construction?*

```
    S
   / \                          /   \
  NP  VP1                      ConnP  VP3
     /   \    /   \             /   \   |
   Barack VP2 Conn  dances
         /  \   |
        smokes and
```

(25) **First, Let’s Figure Out What the Types Have to Be!**

a. \([ [ S ] ] \in D_t\)

b. \([ [ NP ] ] \in D_e\)

c. \([ [ VP_2 ] ] \in D_{<et>}\)

d. \([ [ VP_3 ] ] \in D_{<et>}\)

e. \([ [ VP_1 ] ] \in ??\)

f. \([ [ ConnP ] ] \in ??\)

g. \([ [ and ] ] \in ??\)
(26) **The Type of VP<sub>1</sub>**

- The extension of VP<sub>1</sub> has to combine with the extension of NP (type e) to yield the extension of S (type t)
- So, VP<sub>1</sub> must be of type <et>

(27) **The Type of ConnP**

- The extension of ConnP must combine with the extension of VP<sub>2</sub> (type <et>) to yield the extension of VP<sub>1</sub> (type <et>)
- So, ConnP must be of type <<et><et>>
- *Thus, [[ConnP]] is a function that takes an other function (of type <et>) as argument*

(28) **The Type of Conj**

- The extension of “and” must combine with the extension of VP<sub>3</sub> (type <et>) to yield the extension of ConnP (type <<et><et>>)
- So, “and” must be of type <<et><et><et>>
- *Thus, [[and]] is also a function that takes another function (of type <et>) as argument*

(29) **The Whole Tree**

```
    S<et>
     /  \
 NP<e> VP<sub>1</sub> <et>
    /     <et> <et>
Barack VP<sub>2</sub> ConnP <et>,<et>
    /     <et> <et>
  smokes Conn <et>,<et>,<et>
         / <et>,<et>,<et>
       and dances<et>
```

**OK… we’ve figured out the type of its extension… but what is its exact lexical entry?**
(30) **Targeted Truth-Conditional Statement**

“Barack smokes and dances” is T iff Barack smokes and Barack dances.

With this T-conditional statement, we can now figure out what the extension of VP₁ “smokes and dances” should be!...

(31) **Reasoning Out the Extension of VP₁**

a. **Consider the T-Conditions of Other Sentences Containing VP₁**

   i. \([\text{[Seth smokes and dances]}] = T \text{ iff Seth smokes and Seth dances.}\)
   ii. \([\text{[Joe smokes and dances]}] = T \text{ iff Joe smokes and Joe dances.}\)

b. **Key Generalization**

   \([\text{[[NAME smokes and dances]]} = T \text{ iff NAMED smokes and NAMED dances.}\]

c. **Key Result of Our Type Assignments in (29)**

   \([\text{[[NAME smokes and dances]]} = [\text{[smokes and dances]][[\text{NAME}}]])\]

d. **Recasting our ‘Key Generalization’**

   \([\text{[smokes and dances]]([[\text{NAME}}]) = T \text{ iff NAMED smokes and NAMED dances}\]

   \([\text{[smokes and dances]](\text{NAMED}) = T \text{ iff NAMED smokes and NAMED dances}\]

e. **Key Deduction**

   Given (31c,d), it follows that the extension of VP₁ is a function that takes an entity x as argument, and returns T iff x smokes and x dances.

(32) **Extension Deduced for VP₁ “smokes and dances”**

\[ \lambda x : x \in D_e . \begin{cases} T & \text{IF } x \text{ smokes and } x \text{ dances} \\ F & \text{else} \end{cases} \]

So, we’ve deduced the extension of the VP “smokes and dances”...

And, now that we know the identity of \([\text{[smokes and dances]}]\), as well as the identity of \([\text{[smokes]}]\)…

…we can use that to deduce the extension of the ConnP “and dances”!!
But, before we start, let us make special note of the following equivalence...

### (33) An Important Equivalence to Know

If x is some entity, then the following equivalence holds.

a. **Important Equivalence:** \( "\text{NAME VP}" \approx \"[[\text{VP}]][\text{NAMED}] = T" \)

b. **Illustrations**

(i) Barack smokes \( \approx \[[\text{smokes}]\](Barack) = T \)

(ii) Joe dances \( \approx \[[\text{dances}]\](Joe) = T \)

(iii) Seth laughs \( \approx \[[\text{laughs}]\](Seth) = T \)

...Note that this just follows from our assumption that the extension of a VP is always an <et> function...

### (34) Recasting Our Semantics for VP₁

\[ [[\text{smokes and dances}]] = \]
\[ \lambda x : x \in D_e : \text{IF} [[\text{smokes}}](x) = T \text{ and } [[\text{dances}}](x) = T \text{ THEN } T, \text{ ELSE } F \]

### (35) Reasoning Out the Extension of ConnP, Part 1

a. **The Extensions of Some Other VPs that Contain ConnP**

(i) \[ [[\text{laughs and dances}]] = \]
\[ \lambda x \in D_e : \text{IF} [[\text{laughs}}](x) = T \text{ and } [[\text{dances}}](x) = T \text{ THEN } T, \text{ ELSE } F \]

(ii) \[ [[\text{drinks and dances}]]\]
\[ \lambda x \in D_e : \text{IF} [[\text{drinks}}](x) = T \text{ and } [[\text{dances}}](x) = T \text{ THEN } T, \text{ ELSE } F \]

b. **Key Generalization**

\[ [[\text{VP [and dances]}]] = \]
\[ \lambda x \in D_e : \text{IF} [[\text{VP}}](x) = T \text{ and } [[\text{dances}}](x) = T \text{ THEN } T, \text{ ELSE } F \]
Reasoning Out the Extension of ConnP, Part 2

a. Key Result of Our Type Assignments in (29)

[[ VP [and dances] ]] = [[and dances]]([[VP]])

b. Recasting Our Key Generalization

[[and dances]]([[VP]]) =

[ \lambda x \in D_e : \text{IF } [[VP]](x) = T \text{ and } [[\text{dances}]](x) = T \text{ THEN } T, \text{ ELSE } F ]

c. Key Deduction

(i) The extension of any VP will be some <et> function $f$.

(ii) Thus, given the key generalization in (35b) and (36b), it follows that the extension of the ConnP “and dances” is:

some function that takes an <et> function $f$ and returns
an <et> function that takes an entity $x$ as argument, and yields $T$ iff
$f(x) = T$ and [[dances]](x) = T.

Extension Deduced for the ConnP “and dances”

[ \lambda f : f \in D_{<et>} . [ \lambda x : x \in D_e . \text{IF } f(x) = T \text{ and } [[\text{dances}]](x) = T \text{ THEN } T, \text{ ELSE } F ] ]

So, we’ve deduced the extension of the ConnP “and dances”...

And, now that we know the identity of [[and dances]], as well as the identity of [[dances]]... ...we can use that to deduce the extension of the Conn “and”!!

Reasoning Out the Extension of Conn “And”, Part 1

The Extensions of Some Other ConnPs that Contain “and”

a. [[ and laughs ]] =

[ \lambda f \in D_{<et>} : [ \lambda x \in D_e : \text{IF } f(x) = T \text{ and } [[\text{laughs}]](x) = T \text{ THEN } T, \text{ ELSE } F ] ]

b. [[ and drinks ]] =

[ \lambda f \in D_{<et>} : [ \lambda x \in D_e : \text{IF } f(x) = T \text{ and } [[\text{drinks}]](x) = T \text{ THEN } T, \text{ ELSE } F ] ]
(39) **Reasoning Out the Extension of Conn “And”, Part 2**

a. **Key Generalization**

\[
[ [ \text{and} \ VP ]] = \\
[ \lambda f \in D_{<et>} : [ \lambda x \in D_e : \text{IF } f(x) = T \text{ and } [[VP]](x) = T \text{ THEN } T, \text{ ELSE } F ] ]
\]

b. **Key Result of Our Type Assignments in (29)**

\[
[ [ \text{and} \ VP ]] = [[\text{and}]]([[\text{VP}]])
\]

c. **Recasting Our Key Generalization**

\[
[[\text{and}]]([[\text{VP}]]) = \\
[ \lambda f \in D_{<et>} : [ \lambda x \in D_e : \text{IF } f(x) = T \text{ and } [[VP]](x) = T \text{ THEN } T, \text{ ELSE } F ] ]
\]

d. **Key Deduction**

(i) The extension of any VP will be some <et> function \(g\).

(ii) Thus, given the key generalization in (39a,c), it follows that the extension of the Conn “and” is:

some function that takes an <et> function \(g\) and returns
a function that takes an <et> function \(f\) and returns
and <et> function that takes an entity \(x\) and returns \(T\) iff 
\(f(x) = T \text{ and } g(x) = T\)

(40) **Extension Deduced for the Connective “and” (When Coordinating VPs)**

\[
[ \lambda g \in D_{<et>} : [ \lambda f \in D_{<et>} : [ \lambda x \in D_e : \text{IF } f(x) = T \text{ and } g(x) = T \text{ THEN } T, \text{ ELSE } F ] ] ]
\]

So, we’ve finally deduced the extension of the Conn “and” when it coordinates VPs...

Now, let’s confirm that our theory will work by using the lexical entry in (40) to derive the T-conditions of the sentence “Barack smokes and dances”...
(41) **Sample Derivation of Truth-Conditions**

a. ‘ S ’ is T iff (by notation)

\[
\text{NP} \quad \text{VP}_1 \\
\text{N} \quad \text{VP}_2 \quad \text{Conn} \quad \text{VP}_3 \\
\text{Barack} \quad \text{V}_2 \quad \text{Conn} \quad \text{V}_3 \\
\text{smokes} \quad \text{and} \quad \text{dances}
\]

b. \([S] = T\)

c. **Subproof**

(i) \([\text{NP}] = \) (by NN)

(ii) \([\text{N}] = \) (by NN)

(iii) \([\text{Barack}] = \) (by TN)

(iv) Barack

d. **Subproof**

(i) \([\text{VP}_2] = \) (by NN)

(ii) \([\text{V}_2] = \) (by NN)

(iii) \([\text{smokes}] = \) (by TN)

(iv) \([\lambda x : x \in D_o . \text{IF} x \text{ smokes THEN} T, \text{ELSE} F \])

e. **Subproof**

(i) \([\text{VP}_3] = \) (by NN)

(ii) \([\text{V}_3] = \) (by NN)

(iii) \([\text{dances}] = \) (by TN)

(iv) \([\lambda x : x \in D_o . \text{IF} x \text{ dances THEN} T, \text{ELSE} F \])
f. **Subproof**

(i) \([\text{[Conn]}] = \) (by NN)

(ii) \([\text{[and]}] = \) (by NN)

(iii) \([ \lambda g \in D_{<e>} : [ \lambda f \in D_{<e>} : [ \lambda x \in D_e : 
\begin{align*}
\text{IF} & \ f(x) = T \text{ and } g(x) = T \ 	ext{THEN T, ELSE F } ] ] ]
\end{align*}
\) ]

**Subproof**

(i) \([\text{[ConnP]}] = \) (by FA, e, f)

(ii) \([\text{[Conn]}([\text{[VP}_3]])] = \) (by f)

(iii) \([ \lambda g \in D_{<e>} : [ \lambda f \in D_{<e>} : [ \lambda x \in D_e : 
\begin{align*}
\text{IF} & \ f(x) = T \text{ and } g(x) = T \ 	ext{THEN T, ELSE F } ] ] ]([\text{[VP}_3]]) = \) (by LC)

(iv) \([ \lambda f \in D_{<e>} : [ \lambda x \in D_e : 
\begin{align*}
\text{IF} & \ f(x) = T \text{ and } [\text{[VP}_3](x) = T \ 	ext{THEN T, ELSE F } ] ] = \) (by e)

(v) \([ \lambda f \in D_{<e>} : [ \lambda x \in D_e : [ \lambda x : x \in D_e : 
\begin{align*}
\text{IF} & \ f(x) = T \text{ and } [\lambda x : x \in D_e : \text{IF} \ x \text{ dances THEN T, ELSE F } ](x) = T \ 	ext{THEN T, ELSE F } ] ]
\end{align*}
\) ] = (by LC)

(vi) \([ \lambda f \in D_{<e>} : [ \lambda x \in D_e : [ \lambda x : x \in D_e : \text{IF} \ f(x) = T \text{ and } x \text{ dances } \text{THEN T, ELSE F } ] ] = (by LC)

h. **Subproof**

(i) \([\text{[VP}_1]] = \) (by FA, d, g)

(ii) \([\text{[ConnP]}([\text{[VP}_2]])] = \) (by g)

(iii) \([ \lambda f \in D_{<e>} : [ \lambda x \in D_e : 
\begin{align*}
\text{IF} & \ f(x) = T \text{ and } x \text{ dances } \text{THEN T, ELSE F } ]([\text{[VP}_2]]) = \) (by LC)

(iv) \([ \lambda x \in D_e : [ \text{IF} \ [\text{[VP}_2]](x) = T \text{ and } x \text{ dances THEN T, ELSE F } ] ] = (by d)

(v) \([ \lambda x \in D_e : [ \lambda x : x \in D_e : [ \text{IF} \ x \text{ smokes THEN T, ELSE F } ](x) = T \text{ and } x \text{ dances } \text{THEN T, ELSE F } ] \) = (by LC)

(vi) \([ \lambda x \in D_e : [ \text{IF} \ x \text{ smokes and } x \text{ dances THEN T, ELSE F } ] \)
i. \[[S]\] = T iff (by FA, c, h)

j. \[[VP_1]][[[NP]]] = T iff (by c)

k. \[[VP_1]][[[Barack]]] = T iff (by h)

l. \[\lambda x \in D_c : IF x \text{smokes and } x \text{dances THEN } T, \text{ELSE } F](\text{Barack}) = T iff

m. Barack smokes and Barack dances.

3. Syntactic Well-Formedness vs. Semantic Interpretability

Since they will factor implicitly in some of our subsequent argumentation, it would be useful at this point to clarify certain background assumptions we make regarding the ‘modularity’ of our linguistic systems…

42. Potential Issue: The Interpretability of Ill-Formed Structures

The semantic system we’ve developed for English is able to interpret structures which are actually not possible as English sentences.

a. Ill-Formed Structure Our Semantics Can Interpret

![Diagram of S: NP, VP, Barack, NP, Joe, V likes]

43. Question: Is This a Problem?

Should our semantic system interpret all and only those sentences judged as ‘natural’ or ‘acceptable’ by speakers of the language?

a. One School of Thought (e.g. Richard Montague): YES!

b. Another School of Thought (e.g. Noam Chomsky, us): Not necessarily...
The Overarching Perspective Behind Answer (43b)

- There are many reasons why a speaker might judge a structure to be ‘deviant’.
  
  (a) Deviant for reasons of phonology: *my brushes vs. my brushes
  
  (b) Deviant for reasons of morphology: *gived vs. gave
  
  (c) Deviant for reasons of (pure) syntax: *Barack Joe likes

- Ill-formedness at any one of these levels suffices to explain the overall ‘deviance’
  
  (a) * I gave my brushes OK syntax and semantics; BAD phonology
  
  (b) *I gived a book OK phonology and syntax; BAD morphology
  
  (c) *Barack Joe likes OK semantics (interpretable); BAD syntax

Moreover, our theory predicts that there should be structures in English that are syntactically well-formed, but which our semantics cannot interpret…

Syntactically Well-Formed, but Uninterpretable Structures

Consider the following sentence: “Barack laughed Joe.”

(i) Syntactically Well-Formed

We might imagine that our syntax could generate the following structure:

```
S
  NP
  Barack
  VP
  V
  laughed
  NP
  Joe
```

(ii) Semantically Uninterpretable

- What is [[ S ]]?
- PROBLEM: 
  
  [[laughed]] is of type <et>, and [[Joe]] is of type e
  Thus, [[laughed Joe]] is of type t
  Thus, [[laughed Joe]] cannot ‘combine’ with [[Barack]]
  Thus, we cannot compute a value for [[S]]!
(41) **Terminology: ‘Uninterpretable’**

A structure X is uninterpretable iff [[X]] cannot be computed.

With this terminology in place, we can make the empirical claim in (42) below:

(42) **Empirical Claim**

Uninterpretable structures are perceived by speakers to be ‘deviant’

Thus, the deviance of sentences like “Barack laughed Joe” may be due – not to their syntax *per se* – but their *semantics*, to their inability to be assigned any T-conditions…

- Moreover, if true, the fact in (42) may follow from the following *semantic* principle:

(43) **Principle of Interpretability (Semantic Principle)**

All nodes in a phrase structure tree must be interpretable.