Expanding Our Formalism, Part 2 [1]

1. Lambda Notation for Defining Functions

A Practical Concern:

- Most expressions of natural language will have some kind of function as their extension...
- These more complex functions are very awkward to write in our current notation…

(1) **Example: The Extension of "Or"**

[[or]] = $q: D_t \rightarrow D_{<t,t>}$
for every $x \in D_t$, $q(x) = p_x : D_t \rightarrow D_t$
for every $y \in D_t$, $p_x(y) = T$ iff y=T or x=T

(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 $\varphi(x)$

(3) **Examples**

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

(i) $\{ \langle 0,3 \rangle , \langle 1,4 \rangle, \langle 2,5 \rangle, \langle 3,6 \rangle \}$

(ii) $f : \{ 0, 1, 2, 3 \} \rightarrow \{ 3, 4, 5, 6 \}$
for all $x \in \{ 0, 1, 2, 3 \}$, $f(x) = x + 3$

b. $[\lambda x : x \in \{$ Beatles , Rush $\} .$ the drummer for x $]$ =

(i) { <Beatles, Ringo Starr> , <Rush, Neil Peart> }

(ii) g : { Beatles, Rush } → { y: y is a drummer }
for all x ∈ { Beatles, Rush }, g(x) = the drummer for x

[1] These notes are based on the material in Heim & Kratzer (1998: 34-53).

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

[λx : x ∈ D . φ(x)] (a) = the unique y such that <a,y> ∈ [λx : x ∈ D . φ(x)]

= the function '[λx : x ∈ D . φ(x)]' taking a as argument

(5) **Examples**

a. [λx : x ∈ { 0, 1, 2, 3 } . x + 3](2) = 5

b. [λx : x ∈ { Beatles , Rush } . the drummer for x](Rush) = Neil 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'

[λx : x ∈ D . φ(x)](a) = φ(a)

(7) **Examples**

a. [λx : x ∈ { 0, 1, 2, 3 } . x + 3](2) =

2 + 3 =

5

b. [λx : x ∈ { Beatles , Rush } . the drummer for x](Rush) =

the drummer for Rush =

Neil Peart

(8) **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!

(9) **Example** [2]

[λx : x ∈ **N** . [λz : z ∈ **N** . x + z]]

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

(10) **Example**

[λx : x ∈ **N** . [λz : z ∈ **N** . x + z]](3) = (by LC)
[λz : z ∈ **N** . 3 + z]

(11) **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:

'[λx : x ∈ **N** . [λz : z ∈ **N** . x – z]](3)(4)'

b. <u>How You Read The Formula Above:</u>

- The function '[λx : x ∈ **N** . [λz : z ∈ **N** . x – z]](3)' taking (4) as argument.

c. <u>Equation That Follows From (11b)</u>

[λx : x ∈ **N** . [λz : z ∈ **N** . x – z]](3)(4) = [λz : z ∈ **N** . 3 – z](4)

(12) **Simplification of Lambda Expressions: Examples**

a. (i) [λx : x ∈ **N** . [λz : z ∈ **N** . x – z]](3)(4) = (by LC)
(ii) [λz : z ∈ **N** . 3 – z](4) = (by LC)
(iii) 3 – 4 =
(iv) -1

b. (i) [λx : x ∈ **N** . [λy : y ∈ **N** . [λz : z ∈ **N** . (x+y) – z]]](1)(5)(6) =
(ii) [λy : y ∈ **N** . [λz : z ∈ **N** . (1+y) – z]](5)(6) =
(iii) [λz : z ∈ **N** . (1+5) – z](6) =
(iv) (1+5) – 6 =
(v) 6 – 6 =
(vi) 0

[2] To save space, I will write '**N**' for the set { x : x is a whole number greater than 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.[3]

(14) **Lambda Notation, Part 2** [4]

a. *Syntax*: [$\lambda x : x \in D$. **IF** $\varphi(x)$ **THEN** y, **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**

[$\lambda x : x \in D_e$. **IF** x smokes **THEN** T, **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. [[smokes]]

(16) **Another Example**

[$\lambda x : x \in D_e$. **IF** x dances **THEN** T, **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. [[dances]]

(17) **Still Another Example**

[$\lambda x : x \in D_t$. **IF** x = F **THEN** T, **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. [[it is not the case that]]

---

[3] The notation we will use for representing functions like [[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

(18) **Still One More Example**

[λx : x ∈ D_e . **IF** x likes Joe **THEN** T, **ELSE** F] =

a. The function whose domain is D_e, and for all x ∈ D_e, maps x to T *iff* x likes Joe
b. [[likes Joe]]

Key Observation: Given that [[likes Joe]] is the fomula in (18), we now have at our disposal the means for representing [[likes]] in our lambda notation:

(19) **The Key Example**

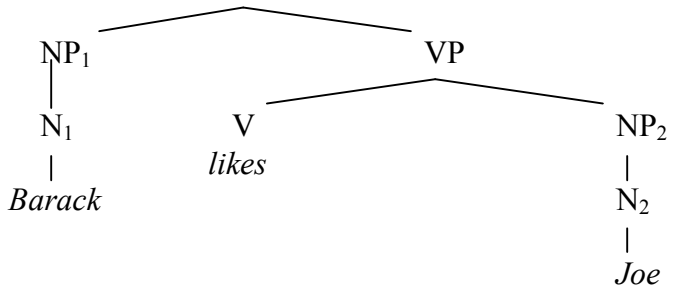
[λy : y ∈ D_e . [λx : x ∈ D_e . **IF** x likes y **THEN** T, **ELSE** F]] =

a. The function whose domain is D_e and for all y ∈ D_e, maps y to…
the function whose domain is D_e, and for all x ∈ D_e, maps x to T *iff* x likes y

b. [[likes]]

(20) **Sample Derivation of Truth-Conditions Using Lambdas**

a. “ S ” is T *iff* (by notation)



b. [[S]] = T

c. **Subproof**

(i) [[NP_1]] = (by NN)

(ii) [[N_1]] = (by NN)

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

(iv) Barack

d. **Subproof**

(i) $[[NP_2]]$ = (by NN)

(ii) $[[N_2]]$ = (by NN)

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

(iv) Joe

e. **Subproof**

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

(ii) [[likes]] = (by TN)

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

f. **Subproof**

(i) [[VP]] = (by FA, d, e)

(ii) $[[V]]([[NP_2]]) =$ (by d)

(iii) [[V]](Joe) = (by e)

(iv) $[\lambda y : y \in D_e . [\lambda x : x \in D_e .$ **IF** x likes y **THEN** T, **ELSE** F]](Joe) = (by LC)

(v) $[\lambda x : x \in D_e .$ **IF** x likes Joe **THEN** T, **ELSE** F]]

g. [[S]] = T *iff* (by FA, c, f)

h. $[[VP]]([[NP_1]]) =$ T *iff* (by c)

i. [[VP]](Barack) = T *iff* (by f)

j. $[\lambda x : x \in D_e .$ **IF** x likes Joe **THEN** T, **ELSE** F](Barack) = T *iff* (by LC) [5]

k. Barack likes Joe.

[5] Note that the derivation of (20k) from (20j) seems on the surface to be more complex than the rule of LC outlined in (6). Recall, however, that our 'if…then' notation in (14) is a 'macro' which can be defined in terms of the simpler notation in (2). Under that definition, the seemingly complex step between (20j) and (20k) does ultimately reduce to multiple steps of LC. Therefore, the use of '(by LC)' as the justification for (20k) is indeed warranted.

(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.

$[\lambda y : y \in D_X . \ldots .]$ = (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**

$[\lambda y_t : [\lambda x_t :$ **IF** x = T or y = T **THEN** T, **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. [[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.

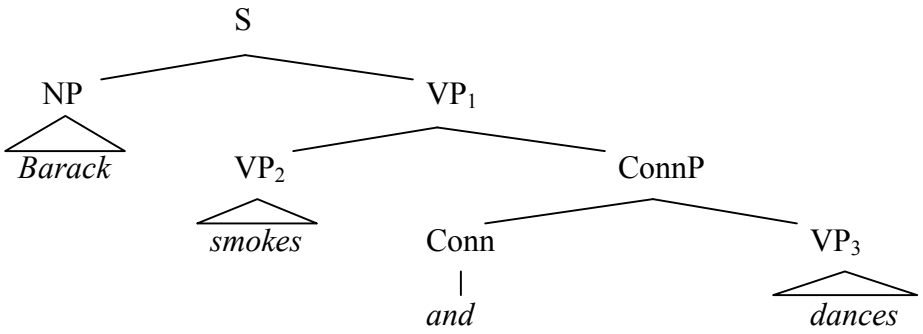
So far, the functions we've constructed have taken either entities or T-values as arguments…
*...but we can also construct functions that take **other functions** as arguments!*

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

$[\lambda f : f \in D_{<et>} . f(\text{Barack})]$ = *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?*



(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_1**

- The extension of VP_1 has to combine with the extension of NP (type e) to yield the extension of S (type t)
- So, VP_1 must be of type <et>

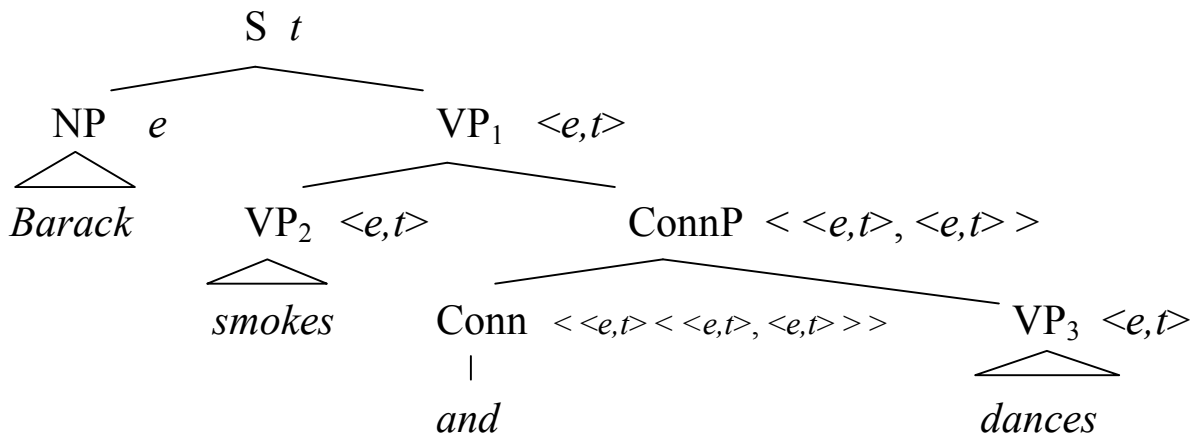
(27) **The Type of ConnP**

- The extension of ConnP must combine with the extension of VP_2 (type <et>) to yield the extension of VP_1 (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_3 (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**



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 VP_1 "smokes and dances" should be!...

(31) **Reasoning Out the Extension of VP_1**

a. Consider the T-Conditions of Other Sentences Containing VP_1

(i) [[Seth smokes and dances]] = T iff Seth smokes and Seth dances.
(ii) [[Joe smokes and dances]] = T iff Joe smokes and Joe dances.

b. Key Generalization

[[**NAME** smokes and dances]] = T iff **NAME** smokes and **NAME** dances.

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

[[**NAME** smokes and dances]] = [[smokes and dances]]([[**NAME**]])

d. Recasting our 'Key Generalization'

[[smokes and dances]]([[**NAME**]]) = T iff **NAME** smokes and **NAME** dances

e. Key Deduction

Given (31c,d), it follows that the extension of VP_1 is a function that takes an entity x as argument, and returns T iff x smokes and x dances.

(32) **Extension Deduced for VP_1 "smokes and dances"**

[$\lambda x : x \in D_e$. **IF** x smokes and x dances **THEN** T, **ELSE** F]

So, we've deduced the extension of the VP "smokes and dances"…

And, now that we know the identity of [[smokes and dances]], as well as the identity of [[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: "x VP" ≈ "[[VP]](x) = T"

b. Illustrations

(i) Barack smokes ≈ [[smokes]](Barack) = T

(ii) Joe dances ≈ [[dances]](Joe) = T

(iii) Seth laughs ≈ [[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 VP_1**

[[smokes and dances]] =

[λx : x ∈ D_e . **IF** [[smokes]](x) = T and [[dances]](x) = T **THEN** T, **ELSE** F]

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

a. <u>The Extensions of Some Other VPs that Contain ConnP</u>

(i) [[laughs and dances]] =
[λx ∈ D_e : **IF** [[laughs]](x) = T and [[dances]](x) = T **THEN** T, **ELSE** F]

(ii) [[drinks and dances]]
[λx ∈ D_e : **IF** [[drinks]](x) = T and [[dances]](x) = T **THEN** T, **ELSE** F]

b. <u>Key Generalization</u>

[[VP [and dances]]] =

[λx ∈ D_e : **IF** [[VP]](x) = T and [[dances]](x) = T **THEN** T, **ELSE** F]

(36) **Reasoning Out the Extension of ConnP, Part 2**

a. <u>Key Result of Our Type Assignments in (29)</u>

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

b. <u>Recasting Our Key Generalization</u>

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

[λx ∈ D_e : **IF** [[VP]](x) = T and [[dances]](x) = T **THEN** T, **ELSE** F]

c. <u>Key Deduction</u>

(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.

(37) **Extension Deduced for the ConnP "and dances"**

[λf : f ∈ $D_{<et>}$. [λx : x ∈ D_e . **IF** f(x) = T and [[dances]](x) = T **THEN** T, **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*"!!

(38) **Reasoning Out the Extension of Conn "And", Part 1**

The Extensions of Some Other ConnPs that Contain "and"

a. [[and laughs]] =
[λf ∈ $D_{<et>}$: [λx ∈ D_e : **IF** f(x) = T and [[laughs]](x) = T **THEN** T, **ELSE** F]]

b. [[and drinks]]
[λf ∈ $D_{<et>}$: [λx ∈ D_e : **IF** f(x) = T and [[drinks]](x) = T **THEN** T, **ELSE** F]]

(39) **Reasoning Out the Extension of Conn "And", Part 2**

a. Key Generalization

[[and VP]] =
[λf ∈ $D_{<et>}$: [λx ∈ D_e : **IF** f(x) = T and [[VP]](x) = T **THEN** T, **ELSE** F]]

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

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

c. Recasting Our Key Generalization

[[and]]([[VP]]) =
[λf ∈ $D_{<et>}$: [λx ∈ D_e : **IF** f(x) = T and [[VP]](x) = T **THEN** T, **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
an <et> function that takes an entity x and returns T *iff*
f(x) = T and *g*(x) = T

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

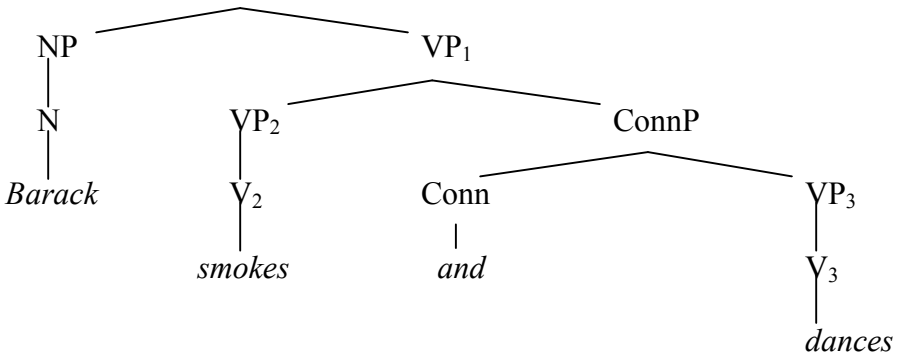
$[\ \lambda g \in D_{<et>} : [\ \lambda f \in D_{<et>} : [\ \lambda x \in D_e$: **IF** f(x) = T and g(x) = T **THEN** T, **ELSE** F]]]

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)



b. [[S]] = T

c. **Subproof**

(i) [[NP]] = (by NN)

(ii) [[N]] = (by NN)

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

(iv) Barack

d. **Subproof**

(i) $[[VP_2]]$ = (by NN)

(ii) $[[V_2]]$ = (by NN)

(iii) [[smokes]] = (by TN)

(iv) $[\ \lambda y : y \in D_e$. **IF** y smokes **THEN** T, **ELSE** F]

e. **Subproof**

(i) $[[VP_3]]$ = (by NN)

(ii) $[[V_3]]$ = (by NN)

(iii) [[dances]] = (by TN)

(iv) $[\ \lambda z : z \in D_e$. **IF** z dances **THEN** T, **ELSE** F]

f. **Subproof**

(i) [[Conn]] = (by NN)

(ii) [[and]] = (by NN)

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

g. **Subproof**

(i) [[ConnP]] = (by FA, e, f)

(ii) $[[Conn]]([[VP_3]])$ = (by f)

(iii) $[\ \lambda g \in D_{<et>} : [\ \lambda f \in D_{<et>} : [\ \lambda x \in D_e :$
IF f(x) = T and g(x) = T **THEN** T, **ELSE** F]]]$([[VP_3]])$ = (by LC)

(iv) $[\ \lambda f \in D_{<et>} : [\ \lambda x \in D_e :$
IF f(x) = T and $[[VP_3]](x)$ = T **THEN** T, **ELSE** F]] = (by e)

(v) $[\ \lambda f \in D_{<et>} : [\ \lambda x \in D_e :$ **IF** f(x) = T and
$[\ \lambda z : z \in D_e$. **IF** z dances **THEN** T, **ELSE** F](x) = T
THEN T, **ELSE** F]] = (by LC)

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

h. **Subproof**

(i) $[[VP_1]]$ = (by FA, d, g)

(ii) $[[ConnP]]([[VP_2]])$ = (by g)

(iii) $[\ \lambda f \in D_{<et>} : [\ \lambda x \in D_e :$
IF f(x) = T and x dances **THEN** T, **ELSE** F]]$([[VP_2]])$ = (by LC)

(iv) [λx ∈ D_e : **IF** [[VP_2]](x) = T and x dances **THEN** T, **ELSE** F] = (by d)

(v) [λx ∈ D_e : **IF** [λy : y ∈ D_e . **IF** y smokes **THEN** T, **ELSE** F](x) = T
and x dances **THEN** T, **ELSE** F]
= (by LC)

(vi) [λx ∈ D_e : **IF** x smokes and x 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. [λx ∈ D_e : **IF** x smokes and x dances **THEN** T, **ELSE** F](Barack) = T *iff* (by LC)

m. Barack smokes and Barack dances.

---

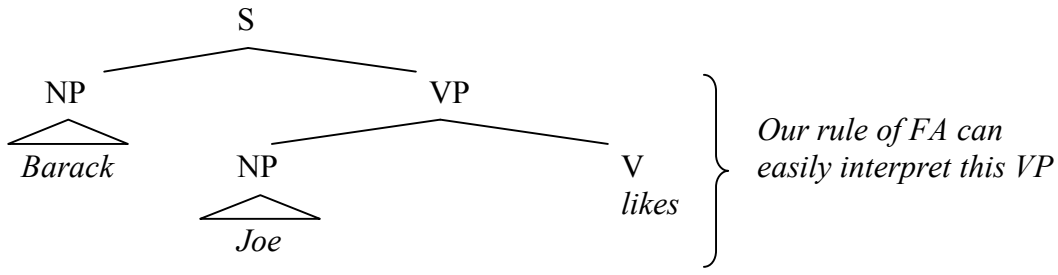
3. Syntactic Well-Formedness vs. Semantic Interpretability

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. <u>Ill-Formed Structure Our Semantics Can Interpret</u>



(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. <u>One School of Thought (*e.g.* Richard Montague):</u> *YES!*

b. <u>Another School of Thought (*e.g.* Noam Chomsky, us):</u> *Not necessarily…*

(44) **The Overarching Perspective Behind Answer (43b)**

- There are many reasons why a speaker might judge a structure to be 'deviant'.

  (a) Deviant purely for reasons of phonology: **my brushs* vs. *my brushes*
  (b) Deviant purely for reasons of morphology: **gived* vs. *gave*
  (c) Deviant purely 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 brushs* 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**

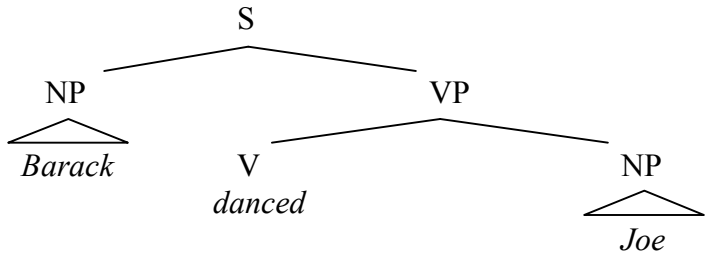
Moreover, our theory predicts that there should be structures in English that are syntactically *well-formed*, but which our semantics cannot interpret…

(45) **Syntactically Well-Formed, but Uninterpretable Structures**

Consider the following sentence: "*Barack danced Joe*."

(i) **The Sentence is (could be) Syntactically Well-Formed**

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



(ii) **The Structure in (i) is Semantically Uninterpretable**

- What is [[S]]?
- PROBLEM:
  [[danced]] is of type <et>, and [[Joe]] is of type e
  Thus, [[danced Joe]] is of type t
  Thus, [[danced 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.

(42) **Empirical Claim**
Uninterpretable structures are perceived by speakers to be 'deviant'

Thus, the deviance of sentences like "*Barack danced 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.

4. A Final Note on the 'Theta-Criterion'

A potential issue with our preceding argument is that, according to certain classic syntactic frameworks, the structure *Barack danced Joe* is actually **syntactically** ill-formed…

…since it violates a **syntactic** condition called 'the Theta-Criterion'…

(44) **Theta Roles (Syntactic Entities)**

Theta roles are a kind of syntactic 'manifestation' of semantic argument structure.

a. Every lexical item is paired with a (possibly null) list of 'theta-roles'.

b. For a lexical item L, there is exactly one 'theta-role' for every argument of [[L]]

c. *Illustration:*

VERB:	*devour*	
THETA-ROLES:	θ_{Agent} ,	θ_{Theme}
	(the eater)	(the eaten)

(45) **Theta-Criterion (Syntactic Principle)**

a. Every NP in a sentence must be 'matched' to exactly one theta-role
b. Every theta-role of lexical item must be 'matched' to exactly one NP

With just this (simplified) background, it's clear that "Barack danced Joe" violates (45)

(46) **Sentence (40b) Violates (45)**

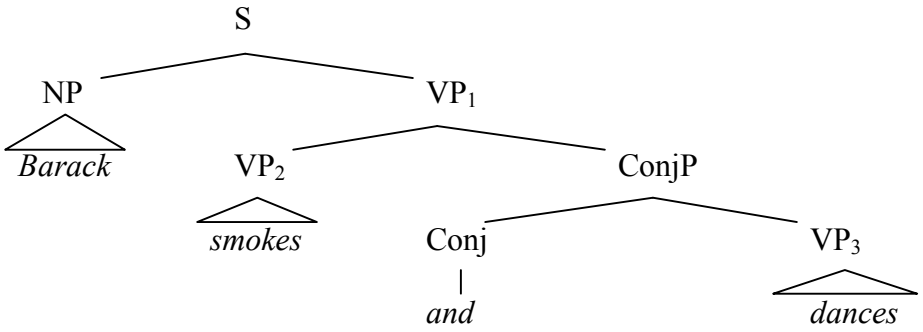
- Since [[danced]] has one argument, *danced* has one theta role
- This one theta role can only be matched to one NP in the sentence (45b)
- Thus, either *Barack* or *Joe* is not matched to a theta-role, in violation of (45a)

Question: *Which is right?*

- *Is (40b) be 'deviant' because it's uninterpretable?*
- *Or is it deviant because it's syntactically ill-formed (violates (45))?*

(47) **A Consideration Against the Theta-Criterion as Stated**

- Recall our analysis of the sentence "Barack smokes and dances". We assumed that it had the following structure:



- If this is the right structure (a 'big *if*'), there is a problem for the 'Theta-Criterion' as stated in (45).
  - There are two theta-roles in the sentence: one from *smokes* and one from *dances*
  - There is only one NP in the sentence: *Barack*
  - Therefore: either one of the theta-roles doesn't get matched to an NP...
    …or the NP *Barack* gets matched to two theta-roles
    (Either way, a violation of (45))

(48) **Conclusion (Tentative / Tendentious)**

We don't seem to lose very much by giving up the 'Theta-Criterion' in (45):

- The structures which (45) is designed to rule out are already predicted to be 'uninterpretable' by our semantics.
- As it's stated in (45), the 'Theta-Criterion' seems to incorrectly rule out certain well-formed (and interpretable) structures of English.

(49) **One Final, Final Note on the Theta-Criterion**

As stated in (45), the 'Theta-Criterion' is also designed to rule out structures like (49a):

a. * Dave devoured.

Our semantics, however, actually does assign an interpretation to (49a)…

> … However, given that "devour" is of type <e<e,t>>, the interpretation assigned to (49a) is an <et> function, not a T-value…
>
> > … and so we correctly predict that (49a) cannot be used to make assertions… (*i.e.* (49a) doesn't have any T-conditions…)