Expanding Our Formalism, Part 2[1]

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"**

[[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

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

(2) **Lambda Notation, Part 1**

a. *Syntax*: [λx : x ∈ D . φ(x)]

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

(3) **Examples**

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

(i) { <0,3> , <1,4>, <2,5>, <3,6> }

(ii) f : { 0, 1, 2, 3 } → { 3, 4, 5, 6 }
for all x ∈ { 0, 1, 2, 3 }, f(x) = x + 3

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

(i) { <Beatles, Ringo Starr> , <Rush, Neal 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: 34-49).

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

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

= 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. [λx : x ∈ { Beatles , Rush } . the drummer for x](Rush) = 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. $[\ \lambda x : x \in \{\ 0, 1, 2, 3\ \}\ .\ x + 3\ ](2) = \mathbf{2 + 3} = 5$

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

$[\ \lambda x : x \in \mathbf{N}\ .\ [\ \lambda z : z \in \mathbf{N}\ .\ x + z\ ]\ ]$ =

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

[2] To save space, I will write ‘**N**’ for the set { x : x is a whole number greater than 0 }

(10) **Example**

$[\ \lambda x : x \in \mathbf{N}\ .\ [\ \lambda z : z \in \mathbf{N}\ .\ x + z\ ]\ ](3)$ = (by LC)

$[\ \lambda z : z \in \mathbf{N}\ .\ 3 + z\ ]$

(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:

'$[\ \lambda x : x \in \mathbf{N}\ .\ [\ \lambda z : z \in \mathbf{N}\ .\ x - z\ ]\ ](3)(4)$'

b. How You Read The Formula Above:

- The function '$[\ \lambda x : x \in \mathbf{N}\ .\ [\ \lambda z : z \in \mathbf{N}\ .\ x - z\ ]\ ](3)$' taking (4) as argument.

c. Equation That Follows From (11b)

$[\ \lambda x : x \in \mathbf{N}\ .\ [\ \lambda z : z \in \mathbf{N}\ .\ x - z\ ]\ ](3)(4) = [\ \lambda z : z \in \mathbf{N}\ .\ 3 - z\ ](4)$

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

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

b. (i) $[\ \lambda x : x \in \mathbf{N}\ .\ [\ \lambda y : y \in \mathbf{N}\ .\ [\ \lambda z : z \in \mathbf{N}\ .\ (x+y) - z\ ]\ ]\ ](1)(5)(6)$ =
(ii) $[\ \lambda y : y \in \mathbf{N}\ .\ [\ \lambda z : z \in \mathbf{N}\ .\ (1+y) - z\ ]\ ](5)(6)$ =
(iii) $[\ \lambda z : z \in \mathbf{N}\ .\ (1+5) - z\ ](6)$ =
(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.[3]

[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.

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

a. *Syntax*: [λx : x $\in$ D . **IF** φ(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* φ(x),
and maps x to z otherwise

(15) **Example**

[λx : 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**

[λx : 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**

[λx : 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]]

(18) **Still One More Example**

[λx : x $\in$ D_e . **IF** x likes Joe **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 likes Joe

b. [[likes Joe]]

---

[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

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:

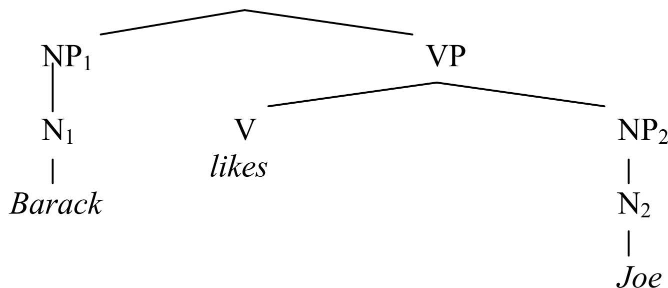
(19) **The Key Example**

[λy : y $\in D_e$. [λx : x $\in D_e$. **IF** x likes y **THEN** T, **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. [[likes]]

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

a. “ 

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

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

[λy : y $\in D_X$. ….] = (i) [λy $\in D_X$: ….]

(ii) [λy_X : ….]

(iii) *when it's clear from context what the domain of the function is, we can even just write:*
[λy : …]

(22) **Example**

[λy_t : [λx_t : **IF** x = T or y = T **THEN** T, **ELSE** F]] =

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

b. [[or]]

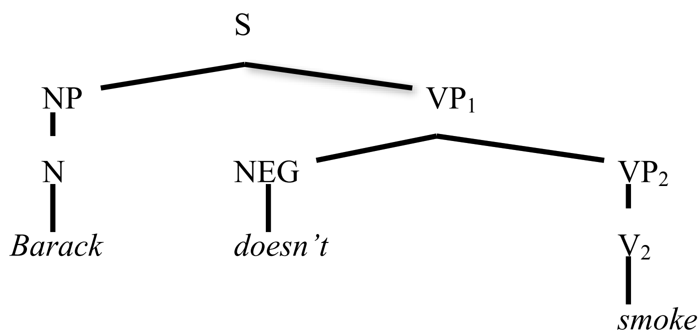
2. Functions That Take Other Functions as Arguments

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) **Extensions that Take Functions as Arguments: Clause-Internal Negation**

- In English, the most natural way to express negation is with the VP modifier *not*
- In a finite clause, '*not*' has to appear with an auxiliary verb, often the 'dummy *do*'
- Let's simplify things a bit, and assume just temporarily that *'doesn't'* is a single head expressing negation…



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

a. [[S]] ∈ D_t b. [[NP]] ∈ D_e

c. [[VP_2]] ∈ $D_{<et>}$ d. **[[VP_1]] ∈ ??**

e. **[[NEG]] ∈ ??** **f.** **[[*doesn't*]] ∈ ??**

(25) **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>

(26) **The Type of NEG**

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

- So, NEG must be of type <<et>,<et>>

- *Thus,* [[NEG]] *is a function that takes an other function (of type <et>) as argument*

(27) **The Type of *Doesn't***

- Since the only mother of *doesn't* in (23) is NEG, it follows from our rule NN that [[*doesn't*]] = [[NEG]]

- Thus, [[*doesn't*]] is also of type <<et>,<et>>

OK… we've figured out the *type* of its extension… but what is its exact lexical entry?

(28) **Targeted Truth-Conditional Statement**

"Barack doesn't smoke" is T *iff* It is not the case that Barack smokes.

With this T-conditional statement, we can now figure out what the extension of VP_1 "doesn't smoke" should be!...

(29) **Reasoning Out the Extension of VP_1 : Part 1**

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

(i) [[Seth doesn't smoke]] = T iff It is not the case that Seth smokes.
(ii) [[Joe doesn't smoke]] = T iff It is not the case that Joe smokes.

b. Key Generalization

[[**NAME** doesn't smoke]] = T iff It is not the case that **NAMED THING** smokes.

Having come up with the generalization in (29b), we can use what we know about the types of the sub-expressions to reason out what the function [[doesn't smoke]] *'does'…*

(30) **Reasoning Out the Extension of VP_1 : Part 2**

a. Key Result of the Type Assignment in (25)

[[**NAME** doesn't smoke]] = [[doesn't smoke]]([[**NAME**]])

b. Trading 'Equals for Equals' in Our Generalization (29b):

[[doesn't smoke]]([[**NAME**]]) = T *iff* It is not the case that **NAMED THING** smokes

c. Using What We Know About the Extensions of Names:

[[doesn't smoke]](**NAMED THING**) = T *iff* It is not the case that **NAMED THING** smokes

d. Key Deduction
Given (30c), it follows that the extension of VP_1 (*doesn't smoke*) is a function that takes an entity x as argument and returns T *iff* it is not the case that x smokes.

(31) **Extension Deduced for VP_1 "doesn't smoke"**

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

- *So, we've deduced the extension of the VP "doesn't somke"…*

- **And, now that we know [[doesn't smoke]], as well as [[smokes]]…**
…we can use that to deduce the extension of [[NEG]]!!!

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: "NAME VP" ≈ "[[VP]](NAMED) = T"

b. Illustrations
(i) Barack smokes ≈ [[smokes]](Barack) = T

(ii) Joe dances ≈ [[dances]](Joe) = T

(iii) Seth laughs ≈ [[laughs]](Seth) = T

(34) **Recasting Our Semantics for VP_1**

[[doesn't smoke]] =

[λx : x ∈ D_e . **IF** it is not the case that [[smokes]](x) = T **THEN** T, **ELSE** F]

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

a. The Extensions of Some Other VPs that Contain NEG / *doesn't:*

(i) [[doesn't laugh]] =
[λx : x ∈ D_e . **IF** it is not the case that [[laughs]](x) = T **THEN** T, **ELSE** F]

(ii) [[doesn't dance]]
[λx : x ∈ D_e . **IF** it is not the case that [[dance]](x) = T **THEN** T, **ELSE** F]

b. Key Generalization
[[doesn't VP]] =
[λx ∈ D_e : **IF** it is not the case that [[VP]](x) = T **THEN** T, **ELSE** F]

Having come up with the generalization in (35b), we can use what we know about the types of the sub-expressions to reason out what the function [[doesn't]] *'does'...*

(36) **Reasoning Out the Extension of NEG**

a. Key Result of the Type Assignment in (26)-(27)

[[doesn't VP]] = [[doesn't]]([[VP]])

b. Trading 'Equals for Equals' in Our Generalization (35b):

[[doesn't]]([[VP]]) =
[λx ∈ D_e : **IF** it is not the case that [[VP]](x) = T **THEN** T, **ELSE** F]

c. Key Deduction

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

(ii) Thus, given the key generalization in (36b), it follows that the extension of the NEG (and *doesn't*) 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*
it is not the case that f(x) = T

(37) **Extension Deduced for the NEG *doesn't***

[λf : f ∈ $D_{<et>}$. [λx : x ∈ D_e . **IF** it is not the case that f(x) = T **THEN** T, **ELSE** F]]

(38) **An Even Simpler Way of Writing Out This Extension**

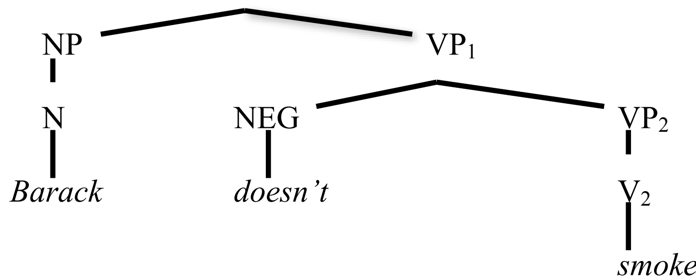
- Note that an <e,t> function only ever give back either T or F.
- Therefore, whenever it is not the case that f(x) = T, it must be that f(x) = F
- Therefore, we can simply our definition of the function in (37) as follows:

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

Now, let's confirm that our theory will work by using the lexical entry in (38) to derive the T-conditions of the sentence "Barack doesn't smoke"…

(39) **Sample Derivation of Truth-Conditions**

a. ' ' 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. **<u>Subproof</u>**

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

(ii) [[*doesn't*]] = (by TN)

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

f. **<u>Subproof</u>**

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

(ii) $[[NEG]]([[VP_2]])$ = (by e)

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

(iv) [$\lambda x : x \in D_e$. **IF** $[[VP_2]]$(x) = F **THEN** T, **ELSE** F] = (by d)

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

(vi) [$\lambda x : x \in D_e$. **IF** it is not the case that x smokes **THEN** T **ELSE** F]

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

h $[[VP_1]]([[NP]])$ = T *iff* (by c)

i. $[[VP_1]]$([[Barack]]) = T *iff* (by f)

j. [$\lambda x: x \in D_e$.
IF it is not the case that x smokes **THEN** T **ELSE** F](Barack) = T *iff* (by LC)

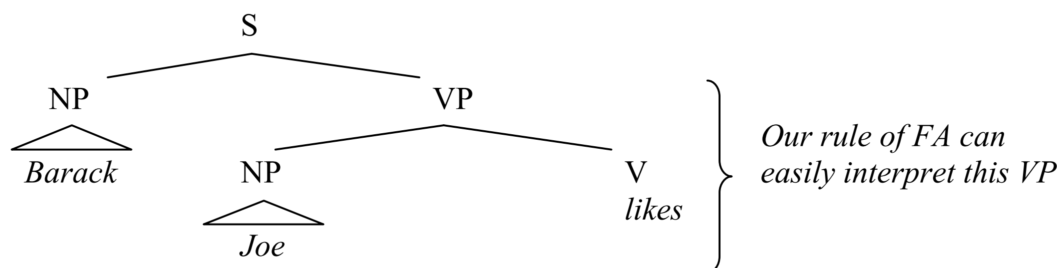
k. It is not the case that Barack smokes

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…

(40) **Potential Issue: The Interpretability of Ill-Formed Structures**
The semantic system we've developed for English is able to interpret structures that are actually not possible as English sentences.

a. Ill-Formed Structure Our Semantics Can Interpret



(41) **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…*

(42) **The Overarching Perspective Behind Answer (41b)**

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

  (a) Deviant for reasons of phonology: **my brushs* 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 brushs* OK syntax and semantics; BAD phonology

  (b) **I gived a book* OK phonology and syntax; BAD morphology

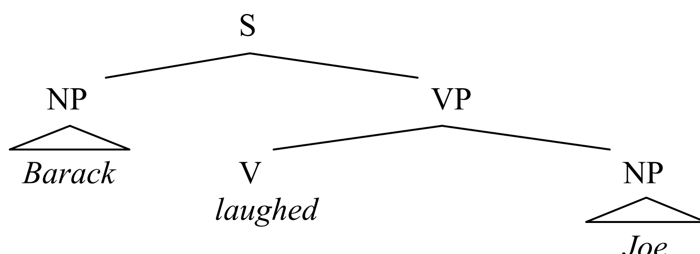
  (c) **Barack Joe likes* OK semantics (interpretable); BAD syntax

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

(43) **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:



(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]]
  <u>Thus, we cannot compute a value for [[S]]!</u>

(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 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.