CHAPTER THREE: LAMBDA-ABSTRACTION

3.1. Syntax and semantics of the λ-operator.

We will now discuss the λ-operator. The syntax of the λ-operator was given as follows:

Functional abstraction:
If x ∈ VAR_a and β ∈ EXP_b then λxβ ∈ EXP_{a,b}

If x is a variable of any type a and β an expression of any type b, then λxβ is an expression of
the type of functions from a-entities into b-entities.

Some examples:

Let x ∈ VAR_e and P ∈ EXP_{<e,t>},
Then (P(x)) ∈ EXP_t.
Given that x ∈ VAR_e and (P(x)) ∈ EXP_t, it follows that:
λx(P(x)) ∈ EXP_{<e,t>}
Given that (P(x)) ∈ EXP_t, also ¬(P(x)) ∈ EXP_t, hence:
λx¬(P(x)) ∈ EXP_{<e,t>}

Let Q ∈ EXP_{<e,t>}. Then (Q(x)) ∈ EXP_t, and
((P(x)) ∧ (Q(x))) ∈ EXP_t. Then:
λx((P(x)) ∧ (Q(x))) ∈ EXP_{<e,t>}

Let P ∈ VAR_{<e,t>}, MARY ∈ CON_e, then (P(MARY)) ∈ EXP_t. Hence:
λP(P(MARY)) ∈ EXP_{<<e,t>,t>}

Let BOY ∈ CON_{<e,t>}, P ∈ VAR_{<e,t>, x ∈ VAR_e}.
Then (BOY(x)) ∈ EXP_t, (P(x)) ∈ EXP_t, hence
((BOY(x)) → (P(x))) ∈ EXP_t.
Thus ∀x((BOY(x)) → (P(x))) ∈ EXP_t. Hence:
λP∀x((BOY(x)) → (P(x))) ∈ EXP_{<<e,t>,t>}

Let Q ∈ VAR_{<e,t>}. Then also ∀x((Q(x)) → (P(x))) ∈ EXP_t.
Hence:
λP∀x((Q(x)) → (P(x))) ∈ EXP_{<<e,t>,t>}

Now, we can apply the rule of forming λ-abstracts to this expression:
Since Q ∈ VAR_{<e,t>, λP∀x((Q(x)) → (P(x))) ∈ EXP_{<<e,t>,t>}, we can abstract over variable
Q:
λQλP∀x((Q(x)) → (P(x))) ∈ EXP_{<<<<e,t>,<e,t>,t>}

Let T,U ∈ VAR_{<<e,t>,t>, P ∈ VAR_{<e,t>},
Then (T(P)) ∈ EXP_t and (U(P)) ∈ EXP_t, hence
((T(P)) ∧ (U(P))) ∈ EXP_t.
Then:
λP((T(P)) ∧ (U(P))) ∈ EXP_{<<e,t>,t>}

50
And: 
$$\lambda T \lambda P ((T(P)) \land (U(P))) \in \text{EXP}_{<<e,t>,t>,<<e,t>,t>>}$$
And: 
$$\lambda U \lambda T \lambda P ((T(P)) \land (U(P))) \in \text{EXP}_{<<e,t>,t>,<<e,t>,t>,<<e,t>,t>>}$$

Let $R,S \in \text{VAR}_{<e,<e,t>>}$, $x,y \in \text{VAR}_e$
Then $((R(y))(x)) \in \text{EXP}_t$ and $((S(y))(x)) \in \text{EXP}_t$.
Hence $[((R(y))(x)) \land ((S(y))(x)) \in \text{EXP}_t$.
Then:
$$\lambda x (((R(y))(x)) \land ((S(y))(x)) \in \text{EXP}_{<e,t>}$$
And: 
$$\lambda y \lambda x (((R(y))(x)) \land ((S(y))(x)) \in \text{EXP}_{<e,<e,t>>}$$
Furthermore:
$$\lambda R \lambda y \lambda x (((R(y))(x)) \land ((S(y))(x)) \in \text{EXP}_{<e,<e,t>},<e,<e,t>>}$$
And: 
$$\lambda S \lambda R \lambda y \lambda x (((R(y))(x)) \land ((S(y))(x)) \in \text{EXP}_{<e,<e,t>},<e,<e,t>},<e,<e,t>$$

We will look at the interpretations of these expressions shortly, but first I will have an excursion on brackets.

**EXCURSION ON BRACKETS**

This is the point where I will give up being very precise about brackets. I will often write $P(x)$, even if in the type logical language I have only introduced $(P(x))$. Similarly, I often write $\phi \land \psi$, even if in the language I have only introduced $(\phi \land \psi)$.

The point about brackets is the following: the logical languages that we introduce are syntactically **unambiguous**: each expression has a unique syntactic derivation. When we define the language, we make sure that this is the case. That's what the brackets are for.

In practice, we are much more sloppy, we leave out brackets, we introduce extra brackets, we use different types of brackets, spacing instead of brackets, and other devices, just to enhance the readability of our expressions.

For example, we write $\phi \land \psi \land \chi$ instead of $((\phi \land \psi) \land \chi)$ or $(\phi \land (\psi \land \chi))$, because it doesn't really matter.

The official rules for brackets in our type logical language -which make sure the language is syntactically unambiguous- are as follows:

-brackets are introduced around all two-place operations.

The two-place operations are: conjunction, disjunction, implication, identity, functional application, hence we write:
$$(\phi \land \psi), (\phi \lor \psi), (\phi \rightarrow \psi), (\alpha=\beta), (\alpha(\beta)).$$
In $(\alpha(\beta))$ the inner brackets are the brackets of the operation of functional application (i.e. they indicate the operation), the outer brackets are the desambiguation brackets.

-one-place operations do not introduce brackets, not to indicate their scope (as in $\neg(\phi)$), nor around them (as in $(\neg \phi)$). This is because for one-place operations, given the other brackets, their scope is predictable, and hence, the whole syntactic structure of, say, $\neg \phi$ is predictable.
The one-place operators are negation, functional abstraction, universal quantification, existential quantification, hence we write: \( \neg \phi, \lambda x \phi, \forall x \phi, \exists x \phi \).

In practice, I will leave out any brackets that I think I can leave out without making the expression unreadable. For readability, I usually choose square brackets to indicate the scope of the quantifiers.

So, I write: \( \forall x [P(x) \rightarrow Q(x)] \) instead of \( \forall x ((P(x)) \rightarrow (Q(x))) \).

Bracketing is important for abstraction and application, because expressions becomes easily unreadable and it becomes easily impossible to determine whether you have first done several abstractions and then several applications, or whether you have done them in different orders.

Look at the following expression:

\[ \lambda z \lambda y \lambda x R(z)(y)(x) \]

It is not clear at all which of the following two syntactic trees this represents:

\[ \lambda z \lambda y \lambda x R(z)(y)(x) \]

a

b

c

\[ z \]

\[ y \]

\[ x \]

\[ R(z)(y)(x) \]

\[ R(z)(y) \]

\[ R(z) \]

\[ R \]

\[ z \]
With all the official rules, we get the following construction trees, that construct different expressions:

(((λzλyλx(((R(z))(y))(x))(c))(b))(a))

(((λzλyλx(((R(z))(y))(x))(c))(b)) a

(λzλyλx(((R(z))(y))(x))(c)) b

λzλyλx(((R(z))(y))(x)) c

z λyλx(((R(z))(y))(x))

y λx(((R(z))(y))(x))

x (((R(z))(y))(x))

((R(z))(y)) x

(R(z)) y

R z
(λz(λy(λx(((R(z))(y))(x))(c))(b))(a))

Since this is difficult to read, we seek a compromise in the notation. For readability, I tend to write a dot after a λ-prefix (a sequence of lambdas), to separate in the functional expression the argument types (indicated by the lambdas) from the function description (indicated by what follows the dot).

The first expression, I would write as:

[λzλyλx.R(z)(y)(x)] (c)(b)(a)

I make the convention that when I write R(z)(y)(x), that is a unit. Hence I assume that λzλyλx.R(z)(y)(x) is built by first applying R to z and to y and to x and then abstracting over x and y and z. The square brackets also indicate that this is a unit, hence we apply to c and to b and to a after abstracting.

Another way of enhancing the readability is to move to relational notation:

[λzλyλx.R(x,y,z)](a,b,c)

This expression represents the same type logical expression.

I further make the convention that in λy.[α](b) the square brackets indicate that the application of α to b takes priority over the abstraction.

I represent the second expression as:

[λz.[λy.[λx.R(z)(y)(x)](c)](b)](a)

Given these conventions, we first apply R to z and y and x, then abstract over x. This applies to c, and then we abstract over y. The result applies to b, and then we abstract over z. The result applies to a. Also here we gain a bit of readability by using relational notation:
[λz.[λy.[λx.R(x,y,z)](c)](b)](a)

But, let me stress, these are my own conventions; other people have different conventions, and you are free to make up your own, as long as the expressions you get are desambiguated enough, i.e. as long as it is clear in your expressions what applies to what.

END OF EXCURS

The semantics of the λ-operator is given as follows:

Functional abstraction.
If \( x \in \text{VAR}_a \) and \( \beta \in \text{EXP}_b \) then:
\[
\llbracket \lambda x \beta \rrbracket_{M,g} = h,
\]
where \( h \) is the unique function in \( (D_a \rightarrow D_b) \) such that:
for every \( d \in D_a \): \( h(d) = \llbracket \beta \rrbracket_{M,g,x,d} \)

I will sometimes use λ-abstraction in the metalanguage and write:
\[
\llbracket \lambda x \beta \rrbracket_{M,g} = \lambda d \in D_a \cdot \llbracket \beta \rrbracket_{M,g,x,d}
\]

This should be understood as just short notation for the definition given before: i.e. \( \lambda d \in D_a \cdot \llbracket \beta \rrbracket_{M,g,x,d} \) should be read as: that function from \( a \)-entities into \( b \)-entities that assigns to every \( d \in D_a \) as value \( \llbracket \beta \rrbracket_{M,g,x,d} \).

Let us look at some examples.

Let \( x \in \text{VAR}_e \) and \( P \in \text{EXP}_{<e,t>} \).
\[
\lambda x.P(x) \in \text{EXP}_{<e,t>}
\]
\[
\llbracket \lambda x.P(x) \rrbracket_{M,g} = h,
\]
where \( h \) is that function from \( D_e \) into \( D_t \), i.e. in \( (D \rightarrow \{0,1\}) \) such that:
for every \( d \in D \): \( h(d) = \llbracket P(x) \rrbracket_{M,g,x,d} \)

This means that \( h \) is that function in \( (D \rightarrow \{0,1\}) \) such that:
for every \( d \in D \): \( h(d) = \llbracket P \rrbracket_{M,g,x,d}(\llbracket x \rrbracket_{M,g,x,d}) \)

i.e. \( h \) is that function in \( (D \rightarrow \{0,1\}) \) such that:
for every \( d \in D \): \( h(d) = F(P)(g^d_x(x)) \)

so \( h \) is that function in \( (D \rightarrow \{0,1\}) \) such that:
for every \( d \in D \): \( h(d) = F(P)(d) \)

This means that for every \( d \in D \): \( h(d) = 1 \) iff \( F(P)(d) = 1 \), and hence that \( h = F(P) \).

Thus:
\[
\llbracket \lambda x.P(x) \rrbracket_{M,g} = F(P) = \llbracket P \rrbracket_{M,g}
\]

It is useful to read variable \( x \) as (generic) 'you'. Then we read the expression \( \lambda x.P(x) \) as:
the property that you have iff you have \( P \)
We see that the semantics of the $\lambda$-operator tells us, as it should, that that property is $P$.

Let $x \in \text{VAR}_e$ and $P \in \text{EXP}_{<e,t>}$. 

\[ \lambda x.\neg P(x) \in \text{EXP}_{<e,t>} \]

\[ \llbracket \lambda x.\neg P(x) \rrbracket_{M,g} = h, \]

where $h$ is the function in $(D \to \{0,1\})$ such that:

for every $d \in D$: $h(d) = \llbracket \neg P(x) \rrbracket_{M,g,d}$

i.e. $h$ is the function in $(D \to \{0,1\})$ such that:

for every $d \in D$: $h(d) = \neg(\llbracket P(x) \rrbracket_{M,g,d})$

Thus $h$ is the function in $(D \to \{0,1\})$ such that:

for every $d \in D$: $h(d) = \neg(\llbracket P \rrbracket_{M,g,d}(\llbracket x \rrbracket_{M,g,d}))$

i.e. $h$ is the function in $(D \to \{0,1\})$ such that:

for every $d \in D$: $h(d) = \neg(F(P)(g^d_x))$

Hence: $h$ is the function in $(D \to \{0,1\})$ such that:

for every $d \in D$: $h(d) = 1$ iff $F(P)(d) = 0$

This means that $\llbracket \lambda x.\neg P(x) \rrbracket_{M,g}$ is the property that you have iff you don’t have $P$.

Thus, if we use constant $\text{WALK} \in \text{CON}_{<e,t>}$ as the representation for $\text{walk}$, then $\lambda x.\neg \text{WALK}(x)$ is an ideal representation for $\text{not walk}$.

Let $P,Q \in \text{EXP}_{<e,t>}$, $x \in \text{VAR}_e$. 

\[ \lambda x. P(x) \land Q(x) \in \text{EXP}_{<e,t>} \]

\[ \llbracket \lambda x. P(x) \land Q(x) \rrbracket_{M,g} = h, \]

where $h$ is the function in $(D \to \{0,1\})$ such that:

for every $d \in D$: $h(d) = \llbracket P(x) \land Q(x) \rrbracket_{M,g,d}$

\[ \llbracket P(x) \land Q(x) \rrbracket_{M,g,d} = \land(\llbracket P(x) \rrbracket_{M,g,d},\llbracket Q(x) \rrbracket_{M,g,d}) \]

\[ = \land(\llbracket P \rrbracket_{M,g,d}(\llbracket x \rrbracket_{M,g,d}),\llbracket Q \rrbracket_{M,g,d}(\llbracket x \rrbracket_{M,g,d})) \]

Hence $h$ is the function in $(D \to \{0,1\})$ such that:

for every $d \in D$: $h(d) = \land(F(P)(d),F(Q)(d))$

Thus, $h$ is the function in $(D \to \{0,1\})$ such that:

for every $d \in D$: $h(d) = 1$ iff $F(P)(d) = 1$ and $F(Q)(d) = 1$
As we have seen, this function is the characteristic function of the set \( \{ d \in D: F(P)(d)=1 \text{ and } F(Q)(d)=1 \} = \{ d \in D: d \in F(P) \text{ and } d \in F(Q) \} = F(P) \cap F(Q) \)

\[ \llbracket \lambda x. P(x) \land Q(x) \rrbracket_{M,g} \]

is the property that you have iff you have both property P and property Q.

Thus, if we choose constants WALK, TALK \( \in \text{CON}_{e,t} \) as representations for \text{walk} and \text{talk},
\[ \lambda x. \text{WALK}(x) \land \text{TALK}(x) \]
represents \text{walk and talk}.

Let P \( \in \text{VAR}_{<e,t>}, \ MARY \in \text{CON}_e \).
\[ \lambda P. P(MARY) \in \text{EXP}_{<e,t,t>} \]

\[ \llbracket \lambda P. P(MARY) \rrbracket_{M,g} = \text{that function } h: ((D \to \{0,1\}) \to \{0,1\}) \text{ such that:} \]
for every K \( \in (D \to \{0,1\}) \): \( h(K) = \llbracket P(MARY) \rrbracket_{M,g,K} \) (note, again, that \( g_{PK} \) is \( g^K \))

= that function h such that:
for every K \( \in D_{<e,t>} \): h(K) = \( g^K P(F(MARY)) \)

= that function h such that:
for every K \( \in D_{<e,t>} \): h(K) = K(F(MARY))

= that function h such that:
for every K \( \in D_{<e,t>} \): h(K)=1 iff K(F(MARY))=1

K is the characteristic function of a set of individuals, set theoretically h is that function such that: for every K \( \in D_{<e,t>} \): h(K)=1 iff F(MARY) \( \in K \)

h itself is the characteristic function of a set of properties (sets), namely:
\{ K \in D_{<e,t>} : h(K)=1 \}

Hence, h characterizes the set: \{ K: F(MARY) \( \in K \) \}: the set of all sets that contain F(MARY), or: the set of all properties that F(MARY) has.

Thus \( \llbracket \lambda P. P(MARY) \rrbracket_{M,g} \) is the set of all properties that Mary has.

Let BOY \( \in \text{CON}_{<e,t>}, P \in \text{VAR}_{<e,t>}, x \in \text{VAR}_e \).
\[ \lambda P. \forall x[\text{BOY}(x) \to P(x)] \in \text{EXP}_{<e,t,t>} \]

\[ \llbracket \lambda P. \forall x[\text{BOY}(x) \to P(x)] \rrbracket_{M,g} = h: D_{<e,t>} \to \{0,1\}, \text{ where} \]
\[ h \text{ is that function from properties into truth values such that:} \]
for every K \( \in D_{<e,t>} \): h(K) = \( \llbracket \forall x[\text{BOY}(x) \to P(x)] \rrbracket_{M,g,K} \)

\[ \llbracket \forall x[\text{BOY}(x) \to P(x)] \rrbracket_{M,g,K} = 1 \text{ iff} \]
for every d \( \in D \): [BOY(x) \( \to P(x) \)]_{M,g,K,d} = 1

This is evaluated relative to the assignment function which is the the result of resetting the
value of P to K and of x to d in g: $g^K_d(x)

iff for every $d \in D$: $F(\text{BOY})(d)=0$ or $K(d)=1$

Hence, h is that function such that: for every $K \in D_{\leq t^2}$:

$$h(K)=1 \text{ iff for every } d \in D: F(\text{BOY})(d)=0 \text{ or } K(d)=1$$

Again, using the set theoretic equivalence:
for every $K$: $h(K)=1 \text{ iff for every } d \in F(\text{BOY})$: $d \in K$

i.e. for every $K$: $h(K)=1$ iff $F(\text{BOY}) \subseteq K$.

Thus, h characterizes the set:

$$\{K \in D_{\leq t^2}: F(\text{BOY}) \subseteq K\}:$$

the set of all sets that $F(\text{BOY})$ is a subset of. This is

$$\{K: \text{ for every } d \in F(\text{BOY}): d \in K\},$$

the set of all properties that every boy has.

Hence $[[\lambda x. \forall x[\text{BOY}(x) \rightarrow P(x)]]_{M,g}$ is the set of all properties that every boy has.

Let $P,Q \in \text{VAR}_{\leq t^2}$, $x \in \text{VAR}_e$.

$$\lambda Q \lambda P. \forall x [Q(x) \rightarrow P(x)] \in \text{EXP}_{\langle e,t^2,\langle e,t^2,t^2 \rangle}$$

$$[[\lambda Q \lambda P. \forall x [Q(x) \rightarrow P(x)]]_{M,g} = h,$n

where h is that function $h: (D_{\leq t^2} \rightarrow D_{\langle e,t^2,t^2 \rangle})$ such that:

For every $L \in D_{\leq t^2}$: $h(L) = [[\lambda P. \forall x [Q(x) \rightarrow P(x)]]_{M,g}QL$

It is easy to see that:

$$[[\lambda P. \forall x [Q(x) \rightarrow P(x)]]_{M,g}QL = \text{that function } j \text{ such that:}$$

for every $K$: $j(K)=1$ iff $L \subseteq K$

Hence, h is that function such that for every $L \in D_{\leq t^2}$, for every $K \in D_{\leq t^2}$:

$$(h(L))(K)=1 \text{ iff } L \subseteq K$$

h characterizes a set of ordered pairs: $\{<K,L>: (h(L))(K)=1\}$, hence h characterizes the set:

$$\{<K,L>: L \subseteq K\}.$$ This is the subset relation, the relation that holds between two sets K and L iff $L \subseteq K$.

Thus $[[\lambda Q \lambda P. \forall x [Q(x) \rightarrow P(x)]]_{M,g}$ is the relation that holds between two sets P and Q iff Q is a subset of P (i.e. iff every Q is a P).

$$\lambda Q \lambda P. \exists x [Q(x) \land P(x)] \in \text{EXP}_{\langle e,t^2,\langle e,t^2,t^2 \rangle}$$

$$[[\lambda Q \lambda P. \exists x [Q(x) \land P(x)]]_{M,g} = h,$n

where h is that function $h: (D_{\leq t^2} \rightarrow D_{\langle e,t^2,t^2 \rangle})$ such that:

For every $L \in D_{\leq t^2}$: $h(L) = [[\lambda P. \exists x [Q(x) \land P(x)]]_{M,g}QL$

$$[[\lambda P. \exists x [Q(x) \land P(x)]]_{M,g}QL = \text{that function } j \text{ such that:}$$

for every $K$: $j(K)=1$ iff for some $d \in D$: $L(d)=1$ and $K(d)=1$
Hence, h is that function such that for every L ∈ D_{e,t} ω, for every K ∈ D_{e,t} ω:
(h(L))(K)=1 iff L ∩ K ≠ Ø

h characterizes \{<K,L>: L ∩ K ≠ Ø \}, the relation that holds between two sets K and L iff L ∩ K ≠ Ø

Thus [\lambda Q.\lambda P. \exists x [Q(x) \land P(x)]]_{M,g} is the relation that holds between two sets P and Q iff the intersection of Q and P is not empty (i.e. iff some Q is a P).

Let A,B ∈ CON_{e,t} and P ∈ VAR_{e,t}.

\lambda P. A(P) \land B(P) ∈ EXP_{e,t}

[\lambda P. A(P) \land B(P)]_{M,g} = h: D_{e,t} ω → \{0,1\}, where

i.e. h is that function such that:
for every K ∈ D_{e,t} ω: h(K) = [\lambda P. A(P) \land B(P)]_{M,g}K

This means that h characterizes the set:
{K: K ∈ F(A) and K ∈ F(B)}, hence h characterizes F(A) ∩ F(B)

[\lambda P. A(P) \land B(P)] is the set of all properties in A ∩ B

Let U,T ∈ VAR_{e,t} and P ∈ VAR_{e,t}.

\lambda U.\lambda T.\lambda P. T(P) \land U(P) ∈ EXP_{e,t}

[\lambda U.\lambda T.\lambda P. T(P) \land U(P)]_{M,g} = h, where

for every B ∈ D_{e,t} ω: h(B) = [\lambda T.\lambda P. T(P) \land U(P)]_{M,g}B

Thus [\lambda U.\lambda T.\lambda P. T(P) \land U(P)]_{M,g} = h, where

for every A, B ∈ D_{e,t} ω: (h(B))(A) = A ∩ B

Thus, using the set theoretic correspondence,

[\lambda U.\lambda T.\lambda P. T(P) \land U(P)]_{M,g} = the function which maps any two sets of sets A and B onto their intersection.

Thus, if \lambda P. P(JOHN) is the representation of the noun phrase John, interpreted as {K: F(JOHN) ∈ K}, the set of all properties that John has, and \lambda P. P(MARY) is the representation of the noun phrase Mary, interpreted as {K: F(MARY) ∈ K}, the set of all properties that Mary has, we can represent noun phrase conjunction as:

\lambda U.\lambda T.\lambda P. T(P) \land U(P). This will give a representation for John and Mary which is interpreted as {K: F(JOHN) ∈ K and F(MARY) ∈ K}, the set of all properties that both John and Mary have.
Let $X, Y \in \text{CON}_{<e,<t,>}$, $x, y \in \text{VAR}_e$

$$\lambda y \lambda x. X(x,y) \wedge Y(x,y) \in \text{EXP}_{<e,<e,t,>}.$$ 

$$[[\lambda y \lambda x. X(x,y) \wedge Y(x,y)]_{M,g}] = h: D \rightarrow D_{<e,t,t,>}$$

where for every $b \in D$: $h(b) = [[\lambda x. X(x,y) \wedge Y(x,y)]_{M,gyb}$

$$[[\lambda x. X(x,y) \wedge Y(x,y)]_{M,gyb}] = j \in D_{<e,t,t,>}$$

where for every $a \in D$: $j(a) = 1 \text{ iff } <a,b> \in F(X) \text{ and } <a,b> \in F(Y)$

Hence $h$ is that function such that:

for every $b \in D$ for every $a \in D$:

$h(b)(a) = 1 \text{ iff } <a,b> \in F(X) \text{ and } <a,b> \in F(Y)$

This means that $h$ characterizes the set $F(X) \cap F(Y)$

Hence $[[\lambda y \lambda x. X(x,y) \wedge Y(x,y)]_{M,g}] = F(X) \cap F(Y)$

Let $R, S \in \text{VAR}_{<e,<e,t,>}$, $x, y \in \text{VAR}_e$

$$\lambda S \lambda R \lambda y \lambda x. R(x,y) \wedge S(x,y) \in \text{EXP}_{<e,<e,t,>,<e,<e,t,>,<e,<e,t,>}>.$$ 

$$[[\lambda S \lambda R y \lambda x. R(x,y) \wedge S(x,y)]_{M,g}]$$

is that function $h$ such that:

for every $B, A \in D_{<e,e,t,t>}$: $(h(B))(A) = A \cap B$

Thus, if KISSED is the representation of kissed and HUGGED the representation of hugged, we can represent transitive verb phrase conjunction and as $\lambda S \lambda R \lambda y \lambda x. R(x,y) \wedge S(x,y)$; its interpretation will take the set of pairs that stand in the kiss relation and the set of pairs that stand in the hug relation, and map them onto the set of pairs that stand both in the kiss and the hug relation.

Let $P \in \text{VAR}_{<e,t,>}$, $x \in \text{VAR}_e$ and $\text{OLD} \in \text{CON}_{<e,t,>}$

$$\lambda P \lambda x. P(x) \wedge \text{OLD}(x) \in \text{EXP}_{<e,e,t,>,<e,t,>}$$

$$[[\lambda P \lambda x. P(x) \wedge \text{OLD}(x)]_{M,g}] = h: (D_e \rightarrow D_i) \rightarrow (D_e \rightarrow D_i)$$

where for every $K \in D_{<e,t,>}$ and for every $d \in D$:

$h(K)(d) = 1 \text{ iff } K(d) = 1 \text{ and } F(\text{OLD})(d) = 1$

i.e. $[[\lambda P \lambda x. P(x) \wedge \text{OLD}(x)]_{M,g}] = h$ such that:

for every $K \in D_{<e,t,>}$ and for every $d \in D$:

$h(K)(d) = 1 \text{ iff } d \in K \text{ and } d \in F(\text{OLD})$

In other words:

$$[[\lambda P \lambda x. P(x) \wedge \text{OLD}(x)]_{M,g}] = h \text{ such that for every } K \in D_{<e,t,>}:
\quad h(K) = K \cap F(\text{OLD})$$

[Note, we are not here dealing with the comparison set dependency of old, we treat it, for ease, as an expression whose comparision is fixed. This is for ease of exposition only.]
Of course, a lot of these expressions (or rather their interpretations) are familiar from the discussion above of sentence (1):

(1) Some old man and every boy kissed and hugged mary.

We gave this the following representation:

\[ \text{SOME(OLD(MAN) AND } \text{1} \text{ EVERY(BOY)) ([KISSED AND } \text{2} \text{ HUGGED](MARY))} \]

This involves choosing the following translation of the lexical items into functional type logic:

- boy → BOY ∈ CON_{<e,t>}
- man → MAN ∈ \text{CON}_{<e,t>}
- mary → MARY ∈ \text{CON}_e
- kissed → KISSED ∈ \text{CON}_{<e,<e,t>}
- hugged → HUGGED ∈ \text{CON}_{<e,<e,t>}
- old → OLD ∈ \text{CON}_{<e,t>,<e,t>}
- every → EVERY ∈ \text{CON}_{<e,t>,<e,t>,<e,t>}
- some → SOME ∈ \text{CON}_{<e,t>,<e,t>,<e,t>}
- and₁ → AND₁ ∈ \text{CON}_{<e,<e,t>,<e,t>,<e,t>,<e,t>}
- and₂ → AND₂ ∈ \text{CON}_{<e,<e,t>,<e,t>,<e,t>,<e,t>,<e,t>,<e,t>}

Then we constrained, through meaning postulates, the meanings of the constants EVERY, SOME, OLD, AND₁ and AND₂.

With the λ-operator, we do not use these constants as the translations of the corresponding lexical items, but we translate them as complex expressions that denote the functions the meaning postulates tell us they should denote (this means, of course, that the meaning postulates become irrelevant):

- old → λPλx.P(x) ∧ OLD(x) ∈ EXP_{<e,t>,<e,t>}
- every → λQλP.∀x[Q(x) → P(x)] ∈ EXP_{<e,t>,<e,t>,<e,t>}
- some → λQλP.∃x[Q(x) ∧ P(x)] ∈ EXP_{<e,t>,<e,t>,<e,t>}
- and₁ → λUλTλP.T(P) ∧ U(P) ∈ EXP_{<e,t>,<e,t>,<e,t>,<e,t>,<e,t>,<e,t>}
- and₂ → λSλRλyλx.R(x,y) ∧ S(x,y) ∈ EXP_{<e,t>,<e,t>,<e,t>,<e,t>,<e,t>,<e,t>,<e,t>}

The advantage of these expressions is that we can read the meaning off the type logical expression:

- λPλx.P(x) ∧ OLD(x) is the function which takes any property P and maps it onto the property λx.P(x) ∧ OLD(x), the property that you have if you have P and you are old.
- λQλP.∀x[Q(x) → P(x)] is the relation that holds between sets P and Q if every Q is a P.
- λQλP.∃x[Q(x) ∧ P(x)] is the relation that holds between sets P and Q if some Q is a P.
- λUλTλP.T(P) ∧ U(P) is the function which takes any two sets of properties A and P and maps them onto λP.U(P) ∧ T(P), the set of properties that are in U and in T.
\[ \lambda S \lambda R \lambda y \lambda x. R(x,y) \land S(x,y) \] is the function which takes two relations R and S and maps them onto the relation \( \lambda y \lambda x. R(x,y) \land S(x,y) \), the relation that x and y stand in iff they stand both in relation R and relation S.

Thus we have gained the advantage of making the representations of the lexical items more perspicuous. One can’t really say that, as such, we have gained a lot of advantage in making the representation of (1) more perspicuous, it now becomes (in infix notation):

\[
\begin{align*}
\end{align*}
\]

Or, undoing infix notation:

\[
\begin{align*}
\end{align*}
\]

This, in fact, seems even worse than:

\[
\begin{align*}
\end{align*}
\]

Yet, there is a big advantage, and that is, that we can use the logical properties of \( \lambda \)-abstraction and functional application to find in a simple way a more readable expression of functional type logic which is logically equivalent to this expression, i.e. we can reduce the representation, by using some rules concerning \( \lambda \)-abstraction. To this we now turn.

3.2. \( \lambda \)-conversion.

We will now discuss some logical properties of the type logical language that we have defined.

In the first place, note that the \( \lambda \)-operator is a variable binding operation, just like the quantifiers (with the difference that it forms functional expressions rather than formulas). This means that some properties of quantifiers as variable binding operations, will be shared by the \( \lambda \)-operator.

In particular, we know that predicate logic satisfies the principle of equivalence of alphabetic variants:

**Principle of equivalence of alphabetic variants for predicate logic:**
For any formula of the form \( Qx \psi \), where \( Qx \) is \( \forall x \) or \( \exists x \), Let \( Qy/\psi[\gamma/x] \) be the result of replacing in \( Qx \psi \), \( Qx \) by \( Qy \) and every occurrence of variable \( x \) which is free in \( \psi \) by variable \( y \). Then:
Qxψ is logically equivalent to Qy/xψ[y/x]

Restrictions: this only holds if the following two conditions are satisfied:
1. Qxψ does not contain an occurrence of variable y which is free in Qxψ (because it would get bound accidentally by the quantifier Qy/x).
2. there is no occurrence of variable x which is free in ψ, such that the occurrence of variable y which is substituted for that occurrence of x in ψ[y/x] is bound in ψ[y/x] (because it would get bound accidentally to the wrong quantifier).

This tells us that ∃y∀x[P(x) → Q(x,y)] is logically equivalent to ∃y∀x[P(x) → Q(x,z)]. These formulas are called alphabetic variants.

The first restriction tells us that ∃x[R(x,y)] and ∃y[R(y,y)] are not alphabetic variants, because, though ∃y[R(y,y)] = ∃y/x[R(x,y)][y/x], variable y was free in ∃x[R(x,y)], but that occurrence of y also gets bound in ∃y[R(x,y)]. Thus condition 1 is violated and there is no guarantee that the equivalence holds.

The second restriction tells us that ∃x∃y[R(x,y)] and ∃y∃y[R(y,y)] are not alphabetic variants. Though ∃y∃y[R(y,y)] = ∃y/x∃y[R(x,y)[y/x]], the occurrence of variable x in ∃x∃y[R(x,y)] is free in ∃y[R(x,y)], but substituting variable y for it, makes both occurrences of y bound to the innermost quantifier ∃y, while this occurrence should be bound to the outermost quantifier.

Note that this principle holds in virtue of the semantics that we have given to predicate logic. When we work out the semantics, we see that ∃y∀x[P(x) → Q(x,y)] is logically equivalent to ∃y∀z[P(z) → Q(z,y)], and similarly, we see that ∃x[R(x,y)] and ∃y[R(y,y)] are not equivalent.

The status of the logical principle of alphabetic variants is that it allows us to reduce a formula to an alphabetic variant, because the semantics tells us that they have the same meaning.

The validity of this principle has only to do with the fact that the quantifiers are variable binding operators. Since the λ-operator is also a variable binding operator, in type logic, this principle holds for the λ-operator as well.

**Principle of identity of alphabetic variants for type logic:**

For any expression of the form Qxψ, where Qx is ∀x or ∃x or λx,

Let Qy/xψ[y/x] be the result of replacing in Qxψ Qx by Qy and every occurrence of variable x which is free in ψ by variable y. Then:

Qxψ = Qy/xψ[y/x] is logically valid (true in all models)

Restrictions: this only holds if the following two conditions are satisfied:
1. Qxψ does not contain an occurrence of variable y which is free in Qxψ (because it would get bound accidentally by the operator Qy/x).
2. there is no occurrence of variable x which is free in ψ, such that the occurrence of variable y which is substituted for that occurrence of x in ψ[y/x] is bound in ψ[y/x] (because it would get bound accidentally to the wrong operator).
This tells us that $\lambda x. P(x) = \lambda y. P(y)$ and that $\lambda P. P(j) = \lambda Q. Q(j)$.

As before, $\lambda y \lambda x. R(x,y) = \lambda z \lambda x. R(x,z)$, but there is no guarantee that $\lambda y \lambda x. R(x,y) = \lambda y \lambda y. R(y,y)$, because this violates the second condition.

Similarly, $\lambda x. \forall y[R(x,y)] = \lambda z. \forall y[R(z,y)]$, but $\lambda x. \forall y[R(x,y)]$ is not guaranteed to be identical to $\lambda y. \forall y[R(y,y)]$, because this violates the second condition.

Again, when we work out the semantics, we see that this indeed holds:

$$\llbracket \lambda x. \forall y[R(x,y)] \rrbracket_{M, g} = h,$$
where for every $d \in D$: $h(d) = 1$ iff $\llbracket \forall y[R(x,y)] \rrbracket_{M, gd} = 1$

= that function $h$ where:
for every $d \in D$: $h(d) = 1$ iff for every $e \in D$: $\llbracket R(x,y) \rrbracket_{M, gxdy} = 1$
= that function $h$ where: for every $d \in D$: $h(d) = 1$ iff for every $e \in D$: $(F(R)(e))(d) = 1$

$$\llbracket \lambda z. \forall y[R(z,y)] \rrbracket_{M, g} = k,$$
where for every $d \in D$: $k(d) = 1$ iff $\llbracket \forall y[R(z,y)] \rrbracket_{M, gd} = 1$

= that function $k$ where:
for every $d \in D$: $k(d) = 1$ iff for every $e \in D$: $\llbracket R(x,y) \rrbracket_{M, gxdy} = 1$
= that function $k$ where: for every $d \in D$: $k(d) = 1$ iff for every $e \in D$: $(F(R)(e))(d) = 1$

Clearly, for every model $M$: $h = k$.

$$\llbracket \lambda y. \forall y[R(y,y)] \rrbracket_{M, g} = j,$$
where for every $d \in D$: $j(d) = 1$ iff $\llbracket \forall y[R(y,y)] \rrbracket_{M, gdy} = 1$

= that function $j$ where:
for every $d \in D$: $j(d) = 1$ iff for every $e \in D$: $\llbracket R(y,y) \rrbracket_{M, gdye} = 1$
= that function $j$ where: for every $d \in D$: $j(d) = 1$ iff for every $e \in D$: $(F(R)(e))(e) = 1$

Clearly, not for every model $M$: $h = j$.

A second logical principle that holds for predicate logic, as we have seen, is extensionality:

**Principle of extensionality for predicate logic:**
Let $\varphi$ be a formula containing an occurrence of term $t$. Let $\varphi[s/t]$ be the result of replacing in $\varphi$ $t$ by $s$.
Let $x_1, ..., x_n$ be the free variables occurring in $s$ and $t$.
Then the following holds:
$$\forall x_1 ... \forall x_n (t = s) \text{ entails } \varphi \leftrightarrow \varphi[s/t]$$
i.e. in every model where $t$ and $s$ have the same extension relative to every assignment function, $\varphi$ and $\varphi[s/t]$ have the same truth value.

Extensionality tells us that if $t$ and $s$ don’t contain free variables then:

$$t = s \text{ entails } \exists z[R(t,z)] \leftrightarrow \exists z[R(s,z)]$$
Type logic is also an extensional logic: the principle of extensionality holds for type logic as well:

**Principle of extensionality for type logic:**
Let \( \varphi \) be a expression containing an occurrence of expression \( \alpha \). Let \( \varphi[\beta/\alpha] \) be the result of replacing in \( \varphi \) \( \alpha \) by \( \beta \).
Let \( x_1, \ldots, x_n \) be the variables free in \( \alpha \) and \( \beta \).
Then the following holds:
\[
\forall x_1, \ldots, \forall x_n (\alpha = \beta) \text{ entails } \varphi = \varphi[\beta/\alpha]
\]

In type logic, this is a very useful principle, which allows us to simplify expressions:

An example:
We know that \( \lambda P.P(j) = \lambda Q.Q(j) \).
Hence \( [\lambda P.P(j)](W) = [\lambda Q.Q(j)](W) \).
We know that \( \lambda x.P(x) \land Q(x) = \lambda x.Q(x) \land P(x) \), hence
\[
[\lambda x.P(x) \land Q(x)](j) = [\lambda x.Q(x) \land P(x)](j).
\]

The principle of extensionality tells us that we can replace within an expression a subexpression by an alphabetic variant, while preserving truth value:

\[
\exists y \forall x[P(x) \rightarrow Q(x,y)] \land R(x,y) \text{ is logically equivalent to } \\
\exists y \forall z[P(z) \rightarrow Q(z,y)] \land R(x,y) \text{ and to } \\
\exists u \forall z[P(z) \rightarrow Q(z,u)] \land R(x,y). \text{ We call the whole expressions also alphabetic variants.}
\]

The central and most useful principle of type logic is the principle of \( \lambda \)-conversion. Let us look at some examples.

Let \( x \in \text{VAR}_e \) and \( P \in \text{CON}_{e,t} \) and \( m \in \text{CON}_e \)
We have already seen above that:
\[
\lambda x.P(x) = P
\]
It follows from extensionality that:
\[
[\lambda x.P(x)](m) = P(m)
\]

We can also show this semantically, of course:
\[
[[\lambda x.P(x)](m)]_{M,\delta} = \\
[\lambda d \in D: [[P(x)]_{M,\text{gxF}(m)}](F(m)) = \\
[[P(x)]_{M,\text{gxF}(m)} \text{ (with assignment } g_\delta^F(m) \text{ )} = \\
([P]_{M,\text{gxF}(m)})(F(m)) = \text{ (since obviously } [[P]_{M,\text{gxF}(m)} = [[P]_{M,\delta}) \\
([P]_{M,\delta})([m]_{M,\delta}) = \\
[P(m)]_{M,\delta}
\]

Hence indeed:
\[
[\lambda x.P(x)](m) = P(m)
\]

Now look at \( [\lambda x.\neg P(x)](m) \).
\[
\llbracket [\lambda x. \neg P(x)](m) \rrbracket_{M,g} =
\]
(\lambda d \in D : \llbracket \neg P(x) \rrbracket_{M,g,d}(F(m)) =
\]
\llbracket \neg P(x) \rrbracket_{M,g,F(m)} =
\neg (\llbracket P(x) \rrbracket_{M,g,F(m)}) = (as we have just seen)
\neg (\llbracket P(m) \rrbracket_{M,g}) =
\llbracket \neg P(m) \rrbracket_{M,g}
\]
Hence:
\[
[\lambda x. \neg P(x)](m) = \neg P(m)
\]

Let moreover \( Q \in \text{CON}_{<e,t>} \)
Look at \([\lambda x. P(x) \land Q(x)](m)\).

\[
\llbracket [\lambda x. P(x) \land Q(x)](m) \rrbracket_{M,g} =
\]
\llbracket [\lambda x. P(x) \land Q(x)] \rrbracket_{M,g}(F(m)) =
\llbracket [\lambda d \in D : [P(x) \land Q(x)]_{M,g,d}(F(m)) =
\]
\llbracket P(x) \land Q(x) \rrbracket_{M,g,F(m)} =
\land (\llbracket P(x) \rrbracket_{M,g,F(m)} \land \llbracket Q(x) \rrbracket_{M,g,F(m)}) =
(by the same reasoning as above)
\land (\llbracket P(m) \rrbracket_{M,g} \land \llbracket Q(m) \rrbracket_{M,g}) =
\llbracket P(m) \land Q(m) \rrbracket_{M,g}
\]
Hence:
\[
[\lambda x. P(x) \land Q(x)](m) = P(m) \land Q(m)
\]

Let us look at another type of example.
Let \( P \in \text{VAR}_{<e,t>}, W \in \text{CON}_{<e,t>}, m \in \text{CON}_e \).

\[
\llbracket [\lambda P. P(m)](W) \rrbracket_{M,g} =
\]
\llbracket [\lambda X \in D_{<e,t>} : [P(m)]_{M,g,F(W)}^X(F(W)) =
\]
\llbracket P(m) \rrbracket_{M,g,F(W)} =
\llbracket P \rrbracket_{M,g,F(W)}(F(m)) =
\llbracket F(W) \rrbracket(F(m)) =
\llbracket W \rrbracket_{M,g}(F(W)) =
\llbracket W(m) \rrbracket_{M,g}
\]
Hence:
\[
[\lambda P. P(m)](W) = W(m)
\]

Next let \( x,y \in \text{VAR}_e, m,j \in \text{CON}_e, R \in \text{CON}_{<e,t>} \)
\[(\lambda y \lambda x. R(y)(x))(m)\]_{M,g} = \\
\lambda d \in D: \left[\lambda x. R(y)(x)\right]_{M,gy,d}(F(m)) = \\
\left[\lambda x. R(y)(x)\right]_{M,g,y,F(m)} = \\
\lambda d \in D: \left[R(y)(x)\right]_{M,g,F(m),y,d}(\text{with } g_{y,F(m)}^d =) = \\
\lambda d \in D: \left[R\right]_{M,g,F(m),y,d}(\text{with } g_{y,F(m)}^d) = \\
\lambda d \in D: F(R)(F(m))(d) = \\
\left[\lambda x. R(m)(x)\right]_{M,g,x,d} = \\
\left[\lambda y \lambda x. R(y)(x)(m)\right]_{M,g} = \lambda y \lambda x. R(y)(x)(m) = \lambda x. R(m)(x)

It follows with extensionality that:
\[\left[\lambda y \lambda x. R(y)(x)(m)\right](j) = \left[\lambda x. R(m)(x)\right](j)

Further:
\[\left[\lambda x. R(m)(x)\right](j)_{M,g} = \\
\lambda d \in D: \left[R(m)(x)\right]_{M,g,x,d}(F(j)) = \\
\left[R(m)(x)\right]_{M,g,x,d}(\text{with } g_{x,F(j)}^d) = \\
\left[F(R)\right](F(m))(F(j)) = \\
\left[R(m)(j)\right]_{M,g} = \\
\left[\lambda y \lambda x. R(y)(x)(m)(j)\right]_{M,g} = \lambda y \lambda x. R(y)(x)(m)(j) = R(m)(j)

In relational notation:
\[\left[\lambda y \lambda x. R(x,y)\right](j,m) = R(j,m)

In all these cases we see the following:

We have a functional expression of type \(\langle a,b \rangle\) of the form \(\lambda x. \beta\), with \(x \in \text{VAR}_a\) and \(\beta \in \text{EXP}_b\). This functional expression applies to an expression \(\alpha\) of type \(a\) (the type that fits \(\lambda x\)). So we have an expression:

\[\left[\lambda x. \beta\right](\alpha)\]

Let us define: \(\beta[\alpha/x]\) is the result of substituting \(\alpha\) for every free occurrence of variable \(x\) in \(\beta\).

By this we mean that this occurrence is free in \(\beta\). This in turn means that it is bound by \(\lambda x\) in \(\lambda x. \beta\).

The generalization from the above cases is that:

\[\left[\lambda x. \beta\right](\alpha) = \beta[\alpha/x]\]

That is, we can simplify an expression \(\left[\lambda x. \beta\right](\alpha)\) by erasing the outermost \(\lambda\)-operator in \(\lambda x. \beta\) and replacing every free occurrence of \(x\) in \(\beta\) by \(\alpha\). This process is called \(\lambda\)-conversion. We
have shown in the above cases that the result of $\lambda$-conversion gives an expression with the same meaning as the original.

The principle of $\lambda$-conversion holds generally for type logic, but, as with the previous logical principles we discussed, there is a restriction concerning illegal binding of variables on it.

Look at the following case:

$$\lambda x. \forall y [R(x,y)](y) \text{ and } \forall y [R(y,y)]$$

$$[[\lambda x. \forall y [R(x,y)]](y)]_{M,g} =$$

$$[\lambda d \in D: [[\forall y [R(x,y)]]_{M, gx d}](g(y)) =$$

$$[[\forall y [R(x,y)]]_{M, gx y}](\text{with } gs^{g(y)})$$

$$[[\forall y [R(x,y)]]]_{M, gx y} = 1 \text{ iff}$$

for every $d \in D: [[R(x,y)]_{M, gx y d}](\text{with } gs^{g(y)} d) = 1 \text{ iff}$$

for every $d \in D: <g(y), d> \in F(R)$

$$[[\forall y [R(y,y)]]]_{M,g} = 1 \text{ iff}$$

for every $d \in D: [[R(y,y)]_{M, gy d}] = 1 \text{ iff}$$

for every $d \in D: <d, d> \in F(R)$

We see that these two expressions do not have the same meaning.

Yet $\forall y [R(y,y)] = \forall y [R(x,y)] [y/x]$, hence $\forall y [R(y,y)]$ is the result of $\lambda$-conversion on $[\lambda x. \forall y [R(x,y)]](y)$. We see, that $\lambda$-conversion does not hold, because variable $y$ was free in the original expression, but got bound by $\forall y$ in the process of conversion. What holds is the following:

**Principle of $\lambda$-conversion for type logic:**

Let $x \in \text{VAR}_a$, $\beta \in \text{EXP}_b$, $\alpha \in \text{EXP}_a$.

Let $[\beta[x/a]]$ be the result of replacing every free occurrence of $x$ in $\beta$ by $\alpha$. Then:

$$[\lambda x. \beta](\alpha) = [\beta[x/a]]$$

if no variable which is free in $\alpha$ gets bound in $[\beta[x/a]]$.

The principle of $\lambda$-conversion tells us that it doesn't matter how complicated our expressions $\lambda x. \beta$ and $\alpha$ are: if $x$ and $\alpha$ are of the same type, and $\alpha$ doesn't contain a variable which is free in $\alpha$ but bound in $[\beta[x/a]]$, then we can reduce the expression $[\lambda x. \beta](\alpha)$ by deleting the $\lambda x$ and substituting $\alpha$ for every occurrence of $x$ that is free in $\beta$, because the expression before conversion and after conversion have provably exactly the same meaning.

We have seen several instances of this reduction above.

We have also seen one case where the reduction doesn't hold:

We cannot reduce $[\lambda x. \forall y [R(x,y)]](y)$ to $\forall y [R(y,y)]$ because these expressions do not have the same meaning.

We can still reduce $\lambda x. \forall y [R(x,y)](y)$ in the following way:

We can take an alphabetic variant of this expression: replace the quantifier $\forall y$ plus the variable that it binds by a different quantifier $\forall z$ binding variable $z$: the result, we have seen,
does have the same meaning:

$$[\lambda x. \forall z[R(x,z)]](y)$$

In this expression we can do $\lambda$-conversion, because now variable $y$ stays free in $\forall z[R(x,z)]$ $[z/x]$.

Hence:

$$[\lambda x. \forall y[R(x,y)]](y) = [\lambda x. \forall z[R(x,z)]](y) = \forall z[R(y,z)]$$

The principle of extensionality tells us that we can do $\lambda$-conversion in a complex expression. We have to be a bit careful here.

Assume that $x,y,z \in \text{VAR}$, $R \in \text{CON}_{<e,<e,<e,t>>}$, $a,b,c \in \text{CON}_e$.

Look at:

$$[\lambda y.[\lambda x.[\lambda z.R(x,y,z)](c)](b)](a)$$

To see what $\lambda$-conversions are possible, we have to look carefully at the syntax of the expression and see which well defined subexpressions of this expression are of the form $[\lambda x. \beta](\alpha)$

There are three such subexpressions:

a.  $[\lambda z.[\lambda y.[\lambda x.R(x,y,z)](a)](b)](c)$
    $[\lambda z. \beta ](\alpha)$

b.  $[\lambda z.[\lambda y.[\lambda x.R(x,y,z)](a)](b)](c)$
    $[\lambda y. \beta ](\alpha)$

c.  $[\lambda z.[\lambda y.[\lambda x.R(x,y,z)](a)](b)](c)$
    $[\lambda x. \beta ](\alpha)$

In all these cases, $\lambda$-conversion is allowed, no variable gets improperly bound. Case a is of course the kind of case that we have seen before. In cases b and c we can do $\lambda$-conversion as well, while preserving meaning, because of the principle of extensionality: we can replace in a complex expression an expression with the same meaning by any other expression with the same meaning, and the $\lambda$-expression and the result of conversion have the same meaning. This means in practice that if in an expression there is more than one sub-expression on which we could do $\lambda$-conversion, which $\lambda$-conversion we do first does not matter: i.e. the order in which we do $\lambda$-conversions does not matter, the result will always be the same.

Thus we can first do the conversion in a, which gives d:

a.  $[\lambda z[\lambda y.[\lambda x.R(x,y,z)](a)](b)](c)$

d.  $[\lambda y.[\lambda x.R(x,y,c)](a)](b)$

Or we can do the conversion in b, which gives e:
e. \([\lambda z. [\lambda x. R(x, b, y)](a)](c)\)

Or the one in c, which gives f:

f. \([\lambda z. [\lambda y. R(a, y, z)](b)](c)\)

All these expressions a...f have the same meaning.

In expression d, we again have two wellformed sub-expressions of the right form for \(\lambda\)-conversion, the same for e and f:

d.1 \([\lambda y. [\lambda x. R(x, y, c)](a)](b)\)  
\[\lambda y. \beta\] (α)

d.2 \([\lambda y. [\lambda x. R(x, y, c)](a)](b)\)  
\[\lambda x. \beta\] (α)

e.1 \([\lambda z. [\lambda x. R(x, b, y)](a)](c)\)  
\[\lambda z. \beta\] (α)

e.2 \([\lambda z. [\lambda x. R(x, b, y)](a)](c)\)  
\[\lambda x. \beta\] (α)

f.1 \([\lambda z. [\lambda y. R(a, y, z)](b)](c)\)  
\[\lambda z. \beta\] (α)

f.2 \([\lambda z. [\lambda y. R(a, y, z)](b)](c)\)  
\[\lambda y. \beta\] (α)

Again, all these conversions give the same result:

both d.1 and e.1 give:

\[\lambda x. R(x, b, c)(a)\]

both d.2 and f.1 give:

\[\lambda y. R(a, y, c)(b)\]

both e.2 and f.2 give:

\[\lambda z. R(a, b, z)(c)\]

All of a...i have the same meaning.

Finally, \(\lambda\)-conversion on g, h or i, gives:

j. \(R(a, b, c)\)

So, we see that all of the following have the same meaning:
Note that we can only do λ-conversion on \([\lambda x.\beta](\alpha)\) in an expression if \([\lambda x.\beta](\alpha)\) is a well-formed sub-expression of that expression. For instance, in a, the following is not a proper sub-expression, hence λ-conversion is not allowed (i.e. there is no guarantee that λ-conversion will preserve meaning, as in fact, it doesn't):

\[ [\lambda z.[\lambda y.[\lambda x.R(x,y,z)](a)](b)](c) \]

This conversion would give \([\lambda y.[\lambda x.R(x,y,a)](b)](c)\).

We see that the a gets to fill the wrong argument place. Obviously this is not equivalent. The reason that λ-conversion is not allowed, is that \([\lambda z.[\lambda y.[\lambda x.R(x,y,z)](a)](b)](c)\) is not a well-formed expression, hence, not a sub-expression of (a.)

Finally, let me mention one more principle that holds in type logic:

**principle of function-identity:**
Let \(\alpha \in \text{EXP}_{a,b}\), \(x \in \text{VAR}_a\). Then
\(\alpha = \lambda x.\alpha(x)\)
if \(\alpha\) doesn't contain a free occurrence of variable \(x\).

3.3. A little grammar.

Let us now make a little grammar that will generate sentence (1) and provide a compositional semantics for it.

As explained before, we will generate syntactic structures, and associate with every syntactic structure a corresponding representation in our type logical language, its translation into type logic. We will do this in a compositional way, that is, we will interpret all the basic expressions, and translate every syntactic operation on syntactic expressions into a semantic operation: the syntactic operation maps input syntactic structures onto an output syntactic structure, the corresponding semantic operation maps the translations of the input syntactic structures onto the translation of the output syntactic expression. Since we have already specified the semantic interpretation of the type logical language, in this way, we associate indirectly with each syntactic structure a corresponding semantic interpretation, namely the interpretation of the corresponding type logical translation.

The grammar will generate pairs \(<\alpha,\beta>\), where \(\alpha\) is a syntactic tree and \(\beta\) is an expression of type logic, the translation of tree \(\alpha\).

We first specify the syntactic categories used in this grammar and their corresponding
semantic types. The grammar will translate any tree with category A as topnode into a type logical expression of the corresponding type. As we will see, in this little grammar certain categories (NP) have more than one corresponding semantic type.

**CATEGORIES AND CORRESPONDING TYPES**

<table>
<thead>
<tr>
<th>Category</th>
<th>Corresponding Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>(&lt;e,t&gt;)</td>
</tr>
<tr>
<td>nouns</td>
<td>sets</td>
</tr>
<tr>
<td>ADJ</td>
<td>(&lt;&lt;e,t&gt;,&lt;e,t&gt;&gt;)</td>
</tr>
<tr>
<td>adjectives</td>
<td>functions from sets into sets</td>
</tr>
<tr>
<td>DET</td>
<td>(&lt;&lt;e,t&gt;,&lt;e,t&gt;,t&gt;&gt;)</td>
</tr>
<tr>
<td>determiners</td>
<td>relations between sets</td>
</tr>
<tr>
<td>NP</td>
<td>(e) or (&lt;&lt;e,t&gt;,t&gt;&gt;)</td>
</tr>
<tr>
<td>noun phrases</td>
<td>individuals or sets of sets</td>
</tr>
<tr>
<td>TV</td>
<td>(&lt;e, &lt;e,t&gt;&gt;)</td>
</tr>
<tr>
<td>transitive verb phrases</td>
<td>two-place relations</td>
</tr>
<tr>
<td>VP</td>
<td>(&lt;e,t&gt;)</td>
</tr>
<tr>
<td>intransitive verb phrases</td>
<td>sets</td>
</tr>
<tr>
<td>S</td>
<td>(t)</td>
</tr>
<tr>
<td>sentences</td>
<td>truth values</td>
</tr>
<tr>
<td>CON_{TV}</td>
<td>(&lt;&lt;e, &lt;e,t&gt;&gt;, &lt;e, &lt;e,t&gt;&gt;, &lt;e, &lt;e,t&gt;&gt;)</td>
</tr>
<tr>
<td>transitive verb phrase connectives</td>
<td></td>
</tr>
<tr>
<td>CON_{NP}</td>
<td>(&lt;&lt;e,t&gt;,t&gt;, &lt;&lt;e,t&gt;,t&gt;, &lt;&lt;e,t&gt;,t&gt;&gt;)</td>
</tr>
<tr>
<td>noun phrase connectives</td>
<td></td>
</tr>
</tbody>
</table>

The grammar contains the following lexical items:

**LEXICAL ITEMS**

\(<N, BOY>\) \(BOY \in CON_{e,t}\)

boy

\(<N, MAN>\) \(MAN \in CON_{<e,t}>\)

man

\(<ADJ, \lambda P \lambda x. P(x) \land OLD(x)\) \(x \in VAR_e, P \in VAR_{<e,t>}\)

old

\(<DET, \lambda Q \lambda P. \forall x [Q(x) \rightarrow P(x)]\) \(x \in VAR_e, P, Q \in VAR_{<e,t>}\)

every

\(<DET, \lambda Q \lambda P. \exists x [Q(x) \land P(x)]\) \(x \in VAR_e, P, Q \in VAR_{<e,t>}\)

some
Let us now specify the rules of the grammar. In what follows
< A, A’ > stands for a pair consisting of a syntactic tree with topnode A and translation A’.

**THE SYNTACTIC AND SEMANTIC RULES**

**R1.** \(<\text{DET},\text{DET}’> + <\text{N},\text{N}’> \implies <\text{NP} , (\text{DET}'(\text{N}') ) >\)**

This rule takes a determiner and a noun and forms an NP, the translation of the NP is the result of applying the translation of the determiner to the translation of the noun.

\( \text{DET}'(\text{N}') \in \text{EXP}_{<e,t,t>} \)

**R2.** \(<\text{ADJ},\text{ADJ}’> + <\text{N},\text{N}’> \implies <\text{N} , (\text{ADJ}'(\text{N}') ) >\)**

\( \text{ADJ}'(\text{N}') \in \text{EXP}_{<e,t>} \)

**R3.** \(<\text{TV},\text{TV}’> + <\text{NP},\text{NP}’> \implies <\text{VP} , (\text{TV}'(\text{NP}') ) >\)**

\( \text{TV}'(\text{NP}') \in \text{EXP}_{<e,t,t>} \)

Note that there is an obvious problem with this rule. It works fine for combining *kissed* with the NP *mary*, but the syntax also allows us to combine *kissed* with the NP *every boy*, and the semantics doesn't work for that case because the types don't match: in that case

\( \text{TV}’ \in \text{EXP}_{<e,\langle e,t,t⟩>} \) and \( \text{NP}’ \in \text{EXP}_{<e,\langle e,t,t⟩,t>} \)

hence \( \text{TV}'(\text{NP}') \) is not wellformed.

Since we are only dealing with an example here, we assume that we only use this rule with
NPs that are proper names. We will come back to this problem shortly.

R4. \[ <NP,NP'> + <VP,VP'> \rightarrow S,(NP'(VP')) \]

\( NP'(VP') \in \text{EXP}_t \)

This rule has the inverse problem from the previous one. This time the rule works fine if we combine the NP every boy with the VP walked, because \( NP' \in \text{EXP}_{<e,t,t>} \) and \( VP' \in \text{EXP}_{<e,t,t>} \). But the syntax allows us to combine mary with walked as well, and in that case \( NP'(VP') \) is not well formed, because \( NP' \in \text{EXP}_e \) and \( VP' \in \text{EXP}_{<e,t,t>} \). Again, we will only be concerned with sentence (1) and assume that the rule doesn’t apply to proper names.

R5. \[ <TV,\alpha'> + <\text{CON}_TV,\text{CON}_TV'> + <TV,\beta'> \rightarrow TV, (\text{CON}_TV'(\beta'))(\alpha') \]

\[ (\text{CON}_TV'(\beta'))(\alpha) \in \text{EXP}_{<e,<e,t,t>} \]

R6. \[ <NP,\alpha'> + <\text{CON}_NP,\text{CON}_NP'> + <NP,\beta'> \rightarrow NP, (\text{CON}_NP'(\beta'))(\alpha') \]

\[ (\text{CON}_NP'(\beta))(\alpha) \in \text{EXP}_{<e,t,t>} \]

This is the grammar. We can now give a derivation for sentence (1):

(1) Some old man and every boy kissed and hugged Mary.

In this derivation, we will see the usefulness of \( \lambda \)-conversion in reducing translations to readable ones.

THE DERIVATION

We start with the lexical items for old and man:

\[ <\text{ADJ}, \lambda P \lambda x. P(x) \land \text{OLD}(x)> \]

\[ \text{old} \]

\[ <\text{N}, \lambda P \lambda x. P(x)> \]

\[ \text{man} \]
R2 applies to these and forms:

\[
< N, [\lambda P \lambda x. P(x) \land OLD(x)](MAN) >
\]

ADJ | N
--- | ---
old | man

The translation is:

\[
[\lambda P \lambda x. P(x) \land OLD(x)](MAN)
\]

This expression can be reduced by \(\lambda\)-converting MAN for P:

\[
[\lambda P \lambda x. P(x) \land OLD(x)](MAN) = \\
\lambda x. MAN(x) \land OLD(x)
\]

the property that you have if you're a man and you're old.

Thus, after reduction, the grammar produces:

\[
< N, \lambda x. MAN(x) \land OLD(x) >
\]

ADJ | N
--- | ---
old | man

Now we take the lexical item for \textit{some}:

\[
< DET, \lambda Q \lambda P. \exists x [Q(x) \land P(x)] >
\]

\[\text{some}\]

and the result we got for \textit{old man}:

\[
< N, \lambda x. MAN(x) \land OLD(x) >
\]

ADJ | N
--- | ---
old | man

Rule R1 applies to these and forms a noun phrase:

\[
< NP, [\lambda Q \lambda P. \exists x [Q(x) \land P(x)]](\lambda x. MAN(x) \land OLD(x)) >
\]

DET | N
--- | ---
some | ADJ | N
old | man
Let us reduce the resulting translation:

\[ [\lambda Q \lambda P. \exists x [Q(x) \land P(x)]] (\lambda x. \text{MAN}(x) \land \text{OLD}(x)) \]

First, just to make the formula more readable, let’s take alphabetic variants and replace \( \lambda x \) by \( \lambda z \):

\[ [\lambda Q \lambda P. \exists x [Q(x) \land P(x)]] (\lambda z. \text{MAN}(z) \land \text{OLD}(z)) = \]

\[ [\lambda Q \lambda P. \exists x [Q(x) \land P(x)]] (\lambda z. \text{MAN}(z) \land \text{OLD}(z)) = \lambda P. \exists x [(\lambda z. \text{MAN}(z) \land \text{OLD}(z))(x) \land P(x)] \]

\[ \lambda z. \text{MAN}(z) \land \text{OLD}(z) \in \text{EXP}_{<e,t>}, \text{ hence we convert it in for } \lambda Q \text{ and it gets substituted for variable } Q: \]

\[ [\lambda Q \lambda P. \exists x [Q(x) \land P(x)]] (\lambda z. \text{MAN}(z) \land \text{OLD}(z)) = \lambda P. \exists x [(\lambda z. \text{MAN}(z) \land \text{OLD}(z))(x) \land P(x)] \]

On this expression, we can do once more \( \lambda \)-conversion, converting variable \( x \) for \( \lambda z \). In this way \( x \) gets substituted for both occurrences of \( z \):

\[ \lambda P. \exists x [(\lambda z. \text{MAN}(z) \land \text{OLD}(z))(x) \land P(x)] = \lambda P. \exists x [(\lambda z. \text{MAN}(z) \land \text{OLD}(z) \land P(x))] \]

the set of properties that some old man has

Thus we get:

\[ <\text{NP} , \lambda P. \exists x [\text{MAN}(x) \land \text{OLD}(x) \land P(x)] > \]

\[ \begin{array}{c}
\text{DET} \\
\text{some ADJ} \\
\text{old man}
\end{array} \]

Next we take the lexical items for every and boy:

\[ <\text{DET} , \lambda Q \lambda P. \forall x [Q(x) \rightarrow P(x)] > \]

\[ \text{every} \]

\[ <\text{N} , \text{BOY} > \]

\[ \text{boy} \]
Once again, rule R1 applies to these and forms a noun phrase:

\[
< \text{NP}, [\lambda Q \lambda P. \forall x [Q(x) \rightarrow P(x)]](\text{BOY}) >
\]

\[
\text{DET} \quad \text{N} \quad \text{every} \quad \text{boy}
\]

We reduce the translation by \(\lambda\)-conversion:

\[
[\lambda Q \lambda P. \forall x [Q(x) \rightarrow P(x)]](\text{BOY}) =
\lambda P. \forall x [\text{BOY}(x) \rightarrow P(x)]
\]

the set of properties that every boy has

Hence we get:

\[
< \text{NP}, \lambda P. \forall x [\text{BOY}(x) \rightarrow P(x)] >
\]

\[
\text{DET} \quad \text{N} \quad \text{every} \quad \text{boy}
\]

We next take *some old man*:

\[
< \text{NP}, \lambda P. \exists x [\text{MAN}(x) \land \text{OLD}(x) \land P(x)] >
\]

\[
\text{DET} \quad \text{N} \quad \text{some} \quad \text{ADJ} \quad \text{N} \quad \text{old} \quad \text{man}
\]

and the lexical item of \textit{and} as an NP-connective:

\[
< \text{CON}_{\text{NP}}, \lambda U \lambda T \lambda P. T(P) \land U(P) >
\]

\[
\text{and}
\]

and *every boy*:

\[
< \text{NP}, \lambda P. \forall x [\text{BOY}(x) \rightarrow P(x)] >
\]

\[
\text{DET} \quad \text{N} \quad \text{every} \quad \text{boy}
\]
And we apply rule R6 to these. This gives:

```
<  NP , [[λUλTλP.T(P) ∨ U(P)][λP.∀x[BOY(x) → P(x)]]
     (λP.∃x[MAN(x) ∧ OLD(x) ∧ P(x)]])

NP  CON  NP
DET N  and  DET N
some ADJ  N  every  boy
old  man
```

Let us reduce the translation:

```
[[λUλTλP.T(P) ∨ U(P)][λP.∀x[BOY(x) → P(x)]]
 (λP.∃x[MAN(x) ∧ OLD(x) ∧ P(x)]]) =

[[λUλTλP.T(P) ∨ U(P)][λQ.∀x[BOY(x) → Q(x)]]
 (λQ.∃x[MAN(x) ∧ OLD(x) ∧ Q(x)]]) = [by λ-conversion]

[λTλP.T(P) ∨ [λQ.∀x[BOY(x) → Q(x)]](P)]
 (λQ.∃x[MAN(x) ∧ OLD(x) ∧ Q(x)])
```

In this expression U is variable of type <<e,t>,t>,
λQ.∀x[BOY(x) → Q(x)] is an expression of type <<e,t>,t>, hence we can apply
λ-conversion. λU disappears, and λQ.∀x[BOY(x) → Q(x)] gets substituted for U in
λTλP.T(P) ∨ U(P). The rest stays as is. So we get:

```
[[λUλTλP.T(P) ∨ U(P)][λQ.∀x[BOY(x) → Q(x)]]
 (λQ.∃x[MAN(x) ∧ OLD(x) ∧ Q(x)]]) = [by λ-conversion]

[λTλP.T(P) ∨ λQ.[∀x[BOY(x) → Q(x)]](P)]
 (λQ.∃x[MAN(x) ∧ OLD(x) ∧ Q(x)])
```

In this expression we can λ-convert P for λQ in [λQ.∀x[BOY(x) → Q(x)]](P), hence we get:

```
[λTλP.T(P) ∨ ∀x[BOY(x) → P(x)]] (λQ.∃x[MAN(x) ∧ OLD(x) ∧ Q(x)])
```

T is a variable of type <<e,t>,t>, hence we can λ-convert
λQ.∃x[MAN(x) ∧ OLD(x) ∧ Q(x)] for λT: λT disappears, and
λQ.∃x[MAN(x) ∧ OLD(x) ∧ Q(x)] gets substituted for T in
λP.T(P) ∨ ∀x[BOY(x) → P(x)]. Hence, we get:
\[ [\lambda T . \lambda P . T(P) \land \forall x [\text{BOY}(x) \rightarrow P(x)]] = (\lambda Q . \exists x [\text{MAN}(x) \land \text{OLD}(x) \land Q(x))] \]

One more \(\lambda\)-conversion converts \(P\) for \(\lambda Q\), giving:

\[ \lambda P . [\lambda Q . \exists x [\text{MAN}(x) \land \text{OLD}(x) \land Q(x)]](P) \land \forall x [\text{BOY}(x) \rightarrow P(x)] = \]

The set of properties that some old man has and that every boy has as well.

Thus we get:

Next we take the lexical items for \text{kissed} and \text{and} as a TV-connective and \text{hugged}:

Rule R5 applies to these and gives:

\[ TV , [[\lambda S \lambda R \lambda y . R(x,y) \land S(x,y)](\text{HUGGED})](\text{KISSED})> \]

S is a variable of type \(<e,<e,t>>\), we convert HUGGED in and get:
\[ ([\lambda S:R \lambda y:R(x,y) \land S(x,y)](HUGGED))(KISSED) = \]
\[ [\lambda R:R(x,y) \land HUGGED(x,y)](KISSED) \]

Converting KISSED for \( \lambda R \) gives:
\[ \lambda y:R(x,y) \land HUGGED(x,y) \]
The relation that me and you stand in if you kissed and hugged me

Thus we get:
\[ < TV, \lambda y:R(x,y) \land HUGGED(x,y)> \]

Next rule R3 combines this and the lexical item for \textit{mary},
\[ < NP, MARY> \]
\[ < VP, [\lambda y:R(x,y) \land HUGGED(x,y)](MARY)> \]
\[ \lambda \text{-conversion converts MARY for } \lambda y, \text{ MARY gets substituted for both occurrences of variable } y: \]
\[ [\lambda y:R(x,y) \land HUGGED(x,y)](MARY) = \]
\[ \lambda x:R(x,\text{MARY}) \land HUGGED(x,\text{MARY}) \]
The property that you have if you kissed Mary and you hugged Mary.

Hence we get:
\[ < VP, \lambda x:R(x,\text{MARY}) \land HUGGED(x,\text{MARY})> \]
Now we take the NP *some old man and every boy* that we have built up:

\[
\langle \text{NP}, \lambda P. \exists x [\text{MAN}(x) \land \text{OLD}(x) \land P(x)] \land \forall x [\text{BOY}(x) \rightarrow P(x)] \rangle
\]

and the VP *kissed and hugged Mary* that we have just built up.

Rule R4 combines the two and gives:

\[
\langle S, \text{NP} \text{CON} \text{NP} \text{TV} \text{CON} \text{NP}, \text{VP} \rangle
\]

Again, to avoid collision of variables we change \( \lambda x \) to \( \lambda z \):

\[
[\lambda P. \exists x [\text{MAN}(x) \land \text{OLD}(x) \land P(x)] \land \forall x [\text{BOY}(x) \rightarrow P(x)]] =
\]

\[
[\lambda P. \exists x [\text{MAN}(x) \land \text{OLD}(x) \land P(x)] \land \forall x [\text{BOY}(x) \rightarrow P(x)]] =
\]

\[
\exists x [\text{MAN}(x) \land \text{OLD}(x) \land [\lambda z. \text{KISSED}(z, \text{MARY}) \land \text{HUGGED}(z, \text{MARY})](x)] \land \\
\forall x [\text{BOY}(x) \rightarrow [\lambda z. \text{KISSED}(z, \text{MARY}) \land \text{HUGGED}(z, \text{MARY})](x)]
\]

There are two \( \lambda \)-conversions left: we convert \( x \) for \( \lambda z \) in the first conjunct, it gets substituted for both occurrences of variable \( z \); and we do the same in the second conjunct. The result is:
There is some man who is old, kissed Mary and hugged Mary and every boy kissed Mary and hugged Mary.

Hence, we finally derive:

\[
\exists x[\text{MAN}(x) \land \text{OLD}(x) \land \text{KISSED}(x,\text{MARY}) \land \text{HUGGED}(x,\text{MARY})] \land \\
\forall x[\text{BOY}(x) \rightarrow \text{KISSED}(x,\text{MARY}) \land \text{HUGGED}(x,\text{MARY})]
\]

The leaves of the syntactic tree form sentence (1), hence the grammar generates that sentence with the above syntactic structure and translation.

Note that at every stage of the derivation, we can semantically interpret the translation of the constituent that the grammar derives at that stage. The translation process, and hence the interpretation, is completely compositional.

Let us define truth for sentences relative to a grammatical analysis:

Let \( \phi \) be a string.

\( <A,A'> \) is a **grammatical analysis of** \( \phi \) iff \( <A,A'> \) is a pair consisting of a syntactic tree with topnode \( A \) and leaves \( \phi \) and \( A' \) is a type logical expression of a type corresponding to \( A \), and the grammar generates \( <A,A'> \).

Let \( \phi \) be a sentence (as a string) and \( <S,S'> \) a grammatical analysis of \( \phi \).

We define:

\( \phi \) is true relative to \( <S,S'> \) in a model \( M \) relative to an assignment function \( g \) iff

\[
[S']_{M,g}=1
\]

Hence we predict that sentence (1) is true in a model

\( M = <D,F> \) relative to an assignment function \( g \) iff:
\[ \exists x [\text{MAN}(x) \land \text{OLD}(x) \land \text{KISSED}(x, \text{MARY}) \land \text{HUGGED}(x, \text{MARY})] \land \\
\forall x [\text{BOY}(x) \rightarrow \text{KISSED}(x, \text{MARY}) \land \text{HUGGED}(x, \text{MARY})] \] 
\[ M_\delta = 1 \text{ iff } \]

for some \( d \in D \): \( d \in F(\text{MAN}) \) and \( d \in F(\text{OLD}) \) and

\[ <d, \text{MARY}> \in F(\text{KISSED}) \text{ and } <d, \text{MARY}> \in F(\text{HUGGED}) \]

and for every \( d \in D \):

\( d \in F(\text{BOY}) \) or \( <d, \text{MARY}> \in F(\text{KISSED}) \) and \( <d, \text{MARY}> \in F(\text{HUGGED}) \)

These are of course the right truth conditions.

In the derivation, I have at every stage done \( \lambda \)-conversions as much as possible. I did not have to do that, of course. Since \( \lambda \)-conversions preserve meaning we can decide at any stage to do or not do a particular \( \lambda \)-conversion. Without doing any \( \lambda \)-conversions, we would have generated sentence (1) with translation:

\[ ([\lambda U \lambda T \lambda P. T(P) \land U(P))]([\lambda Q \lambda P. \exists x [Q(x) \rightarrow P(x)](\text{BOY}) \\
(\lambda Q \lambda P. \exists x [Q(x) \land P(x)](\lambda P \lambda x. P(x) \land \text{OLD}(x)(\text{MAN}))) ] \\
([\lambda S \lambda R \lambda y \lambda x. R(x, y) \land S(x, y)](\text{HUGGED})(\text{KISSED})(\text{MARY})) = \]

In this expression we can do all the \( \lambda \)-conversions that we did above along the way:

\[ ([\lambda U \lambda T \lambda P. T(P) \land U(P))]([\lambda Q \lambda P. \exists x [Q(x) \rightarrow P(x)](\text{BOY}) \\
(\lambda Q \lambda P. \exists x [Q(x) \land P(x)](\lambda P \lambda x. P(x) \land \text{OLD}(x)(\text{MAN}))) ] \\
([\lambda S \lambda R \lambda y \lambda x. R(x, y) \land S(x, y)](\text{HUGGED})(\text{KISSED})(\text{MARY})) = \]

[converting MAN]

\[ ([\lambda U \lambda T \lambda P. T(P) \land U(P))]([\lambda Q \lambda P. \exists x [Q(x) \rightarrow P(x)](\text{BOY}) \\
(\lambda Q \lambda P. \exists x [Q(x) \land P(x)](\lambda x. \text{MAN}(x) \land \text{OLD}(x))) ] \\
([\lambda S \lambda R \lambda y \lambda x. R(x, y) \land S(x, y)](\text{HUGGED})(\text{KISSED})(\text{MARY})) = \]

[converting \( \lambda x. \text{MAN}(x) \land \text{OLD}(x) \)]

\[ ([\lambda U \lambda T \lambda P. T(P) \land U(P))]([\lambda Q \lambda P. \exists x [Q(x) \rightarrow P(x)](\text{BOY}) \\
(\lambda P. \exists x [([\lambda x. \text{MAN}(x) \land \text{OLD}(x)])(x) \land P(x)]) ] \\
([\lambda S \lambda R \lambda y \lambda x. R(x, y) \land S(x, y)](\text{HUGGED})(\text{KISSED})(\text{MARY})) = \]

[converting \( x \)]

\[ ([\lambda U \lambda T \lambda P. T(P) \land U(P))]([\lambda Q \lambda P. \exists x [Q(x) \rightarrow P(x)](\text{BOY}) \\
(\lambda P. \exists x [\text{MAN}(x) \land \text{OLD}(x) \land P(x)]) ] \\
([\lambda S \lambda R \lambda y \lambda x. R(x, y) \land S(x, y)](\text{HUGGED})(\text{KISSED})(\text{MARY})) = \]

[converting BOY]

\[ ([\lambda U \lambda T \lambda P. T(P) \land U(P))]([\lambda P. \forall x [\text{BOY}(x) \rightarrow P(x)]]) \\
(\lambda P. \exists x [\text{MAN}(x) \land \text{OLD}(x) \land P(x)]) ] \\
([\lambda S \lambda R \lambda y \lambda x. R(x, y) \land S(x, y)](\text{HUGGED})(\text{KISSED})(\text{MARY})) = \]

[converting \( \lambda P. \forall x [\text{BOY}(x) \rightarrow P(x)] \)]

\[ ([\lambda T \lambda P. T(P) \land [\lambda P. \forall x [\text{BOY}(x) \rightarrow P(x)]]) ] \\
(\lambda P. \exists x [\text{MAN}(x) \land \text{OLD}(x) \land P(x)]) ] \\
([\lambda S \lambda R \lambda y \lambda x. R(x, y) \land S(x, y)](\text{HUGGED})(\text{KISSED})(\text{MARY})) = \]

[converting P]
[ [(\lambda T \lambda P. T)(P) \land \forall x[BOY(x) \rightarrow P(x)]]
  (\lambda P. \exists x[MAN(x) \land OLD(x) \land P(x)])
  ([(\lambda S \lambda R \lambda y \lambda x. R(x,y) \land S(x,y))(HUGGED)(KISSED)(MARY)] =
   [converting \lambda P. \exists x[MAN(x) \land OLD(x) \land P(x)]]
   [\lambda P. [\lambda P. \exists x[MAN(x) \land OLD(x) \land P(x)](P) \land \forall x[BOY(x) \rightarrow P(x)]]
   ([(\lambda S \lambda R \lambda y \lambda x. R(x,y) \land S(x,y))(HUGGED)(KISSED)(MARY)] =
    [converting P]
   [\lambda P. \exists x[MAN(x) \land OLD(x) \land P(x)] \land \forall x[BOY(x) \rightarrow P(x)]]
   ([(\lambda S \lambda R \lambda y \lambda x. R(x,y) \land S(x,y))(HUGGED)(KISSED)(MARY)] =
    [converting HUGGED]
   [\lambda P. \exists x[MAN(x) \land OLD(x) \land P(x)] \land \forall x[BOY(x) \rightarrow P(x)]]
   ([(\lambda R \lambda y \lambda x. R(x,y) \land HUGGED(x,y))(KISSED)(MARY)] =
    [converting KISSED]
   [\lambda P. \exists x[MAN(x) \land OLD(x) \land P(x)] \land \forall x[BOY(x) \rightarrow P(x)]]
   ([(\lambda y \lambda x. KISSED(x,y) \land HUGGED(x,y))(MARY)] =
    [converting MARY and relational notation]
   [\lambda P. \exists x[MAN(x) \land OLD(x) \land P(x)] \land \forall x[BOY(x) \rightarrow P(x)]]
   (\lambda x. KISSED(x,MARY) \land HUGGED(x,MARY)) =
    [converting \lambda x. KISSED(x,MARY) \land HUGGED(x,MARY)]
   \exists x[MAN(x) \land OLD(x) \land [\lambda x. KISSED(x,MARY) \land HUGGED(x,MARY)](x)] \land
   \forall x[BOY(x) \rightarrow [\lambda x. KISSED(x,MARY) \land HUGGED(x,MARY)](x)] =
    [converting x]
   \exists x[MAN(x) \land OLD(x) \land KISSED(x,MARY) \land HUGGED(x,MARY)] \land
   \forall x[BOY(x) \rightarrow [\lambda x. KISSED(x,MARY) \land HUGGED(x,MARY)](x)] =
    [converting x]
   \exists x[MAN(x) \land OLD(x) \land KISSED(x,MARY) \land HUGGED(x,MARY)] \land
   \forall x[BOY(x) \rightarrow KISSED(x,MARY) \land HUGGED(x,MARY)]
And we get the same result.

We see that the level of translation is a medium to represent the meanings, rather than the meanings themselves: we choose not to do \lambda-conversion along the way, when we want to keep track in the translation which functional expressions we have applied to which argument expressions; we do \lambda-conversion along the way, when we want, at each stage of the derivation have the clearest idea what the meaning derived at that stage actually is. The latter is the more common case. Important is to keep in mind that in this whole process we have derived per constituent only one syntactic tree with only one meaning, because in each reduction all the type logical expressions have the same meaning.