

1. Introduction

As we understand the semantic enterprise, the semantic-analysis of a phrase ϕ consists in the following.

- (1) providing a *semantic-value* for ϕ ;
- (2) providing a *semantic-value* for each of ϕ 's component phrases,
- (3) providing an account of how the semantic-value of ϕ is constructed out of the semantic-values of its component phrases.

We propose to carry out this enterprise using a formal language based on type-theory and lambda-abstraction. This formal language is understood as a target-language some of whose phrases serve as translations of the object-phrases, and in particular have the same syntactic-types as the object phrases.

2. Basic Type-Theoretic Language – TL_0

1. Types

- (1) D is a type [definite-noun-phrases]
- (2) S is a type [sentences]
- (3) if $\mathfrak{I}_0, \dots, \mathfrak{I}_k$ are types, then so is $(\mathfrak{I}_1 \times \dots \times \mathfrak{I}_k) \rightarrow \mathfrak{I}_0$ [k -place functors]
- (4) nothing else is a type.

2. Formation Rules

1. Variables

For each type \mathfrak{I} , there is an infinite list of variables of type \mathfrak{I} .

Note: we employ just a few types of variables, which are type-encoded as follows.

- | | | |
|----------------------------|---|-----------------------------------|
| (1) lower-case math-italic | x, y, z, \dots | D |
| (2) upper-case times-roman | P, Q, R, \dots | $D \rightarrow S$ |
| (3) upper-case block | $\mathbb{P}, \mathbb{Q}, \mathbb{R}, \dots$ | $(D \rightarrow S) \rightarrow S$ |
| (4) upper-case Greek | Φ, Ψ, \dots | S |

2. Proper Expressions

The proper expressions of a language are the special expressions of that language. For example, in arithmetic, the proper expressions of type D are proper names '0', '1', '2', etc., and the proper expressions of type $D \rightarrow S$ are phrases such as 'is even', 'is odd', etc.

On the other hand, in an abstract language, the proper expressions are just letters, as in elementary logic, whose readings are given by an attendant (often tacit) glossary.

We propose an abstract language, using the following typing convention for the proper expressions.

- | | | |
|---------------------|---|---------------------|
| (1) small-caps | J, K, L, \dots | D |
| (2) bold lower-case | $\mathbf{f}, \mathbf{g}, \mathbf{h}, \dots$ | $D^* \rightarrow D$ |
| bold small-caps | $\mathbf{F}, \mathbf{G}, \mathbf{H}, \dots$ | $D^* \rightarrow D$ |
| (3) bold upper-case | $\mathbf{P}, \mathbf{Q}, \mathbf{R}, \dots$ | $D^* \rightarrow S$ |

Here, D^* is a wildcard type, which can be D , $D \times D$, $D \times D \times D$, etc. Note in particular that our proper expressions all have "flat" types in the manner of elementary logic; for example, two-place functors have type $(D \times D) \rightarrow D$.

We also follow elementary logic in writing function-signs using round-parentheses, and writing predicates using square-brackets, which is illustrated in the following glossary.¹

$\mathbf{R}[\alpha, \beta]$	$=:$	α respects β
$\mathbf{f}(\alpha)$	$=:$	α 's father ; the father of α
$\mathbf{m}(\alpha)$	$=:$	α 's mother ; the mother of α
J	$=:$	Jay
K	$=:$	Kay
Thus:		
$\mathbf{R}[\mathbf{f}(\mathbf{J}), \mathbf{m}(\mathbf{K})]$	$=:$	Jay's father respects Kay's mother

3. Connectives

If Φ and Ψ are formulas, then so are the following.²

$$\begin{array}{l} \sim\Phi \\ [\Phi \ \& \ \Psi] \\ [\Phi \ \vee \ \Psi] \\ [\Phi \ \rightarrow \ \Psi] \\ [\Phi \ \leftrightarrow \ \Psi] \end{array}$$

4. Identity

If \mathcal{E}_1 and \mathcal{E}_2 are expressions of the same type, then

$$[\mathcal{E}_1 = \mathcal{E}_2]$$

is a formula.³

5. Quantification

If Φ is a formula, and v is a variable, of any type, then

$$\begin{array}{l} \forall v\Phi \\ \exists v\Phi \end{array}$$

are formulas.

6. Definite-Description

If v is a variable of type \mathfrak{I} , and Φ is a formula, then

$${}_1v\Phi$$

is an expression of type \mathfrak{I} .

7. Lambda-Abstraction

If v_1, \dots, v_k are distinct variables of types $\mathfrak{I}_1, \dots, \mathfrak{I}_k$, respectively, and \mathcal{E} is an expression of type \mathfrak{I}_0 , then

$$\lambda v_1 \dots v_k \mathcal{E}$$

is an expression of type $(\mathfrak{I}_1 \times \dots \times \mathfrak{I}_k) \rightarrow \mathfrak{I}_0$.

¹ We drop brackets for predicates when all its arguments are simple, but we never drop parentheses for function-signs.

² The outer brackets admit variant spellings, and also are usually dropped when the formula stands alone.

³ See footnote 2.

8. Functor-Argument Compounds

If ϕ is a functor of type $(\mathfrak{I}_1 \times \dots \times \mathfrak{I}_k) \rightarrow \mathfrak{I}_0$,
and $\alpha_1, \dots, \alpha_k$ are expressions of types $\mathfrak{I}_1, \dots, \mathfrak{I}_k$, respectively, then

$$[\phi]\langle \alpha_1, \dots, \alpha_k \rangle$$

is an expression of type \mathfrak{I}_0 .

9. Variant Notation

We use variant bracket-notation, including dropping brackets, in a manner that is consonant with elementary logic. We also adopt the following left-association rule for square-brackets.

$$[\Lambda]\langle \alpha_1, \dots, \alpha_m \rangle \langle \beta_1, \dots, \beta_n \rangle \quad =_{df} \quad [[\Lambda]\langle \alpha_1, \dots, \alpha_m \rangle] \langle \beta_1, \dots, \beta_n \rangle$$

10. Order

$order(D)$	$=_{df}$	0
$order(S)$	$=_{df}$	0
$order(\mathcal{A} \rightarrow \mathcal{B})$	$=_{df}$	$max \{ 1 + order(\mathcal{A}), order(\mathcal{B}) \}$

Examples:

	=		Dot-Notation
$order[D \rightarrow S]$	=	1	$D \rightarrow S$
$order[D \rightarrow (D \rightarrow S)]$	=	1	$D \rightarrow D \rightarrow S$
$order[D \rightarrow (D \rightarrow (D \rightarrow S))]$	=	1	$D \rightarrow D \rightarrow D \rightarrow S$
$order[(D \rightarrow S) \rightarrow S]$	=	2	$D \rightarrow S. \rightarrow S$
$order[(D \rightarrow S) \rightarrow (D \rightarrow S)]$	=	2	$D \rightarrow S. \rightarrow D \rightarrow S$
$order[((D \rightarrow S) \rightarrow S) \rightarrow S]$	=	3	$D \rightarrow S. \rightarrow S. \rightarrow S$
$order[(((D \rightarrow S) \rightarrow S) \rightarrow S) \rightarrow S]$	=	4	$D \rightarrow S. \rightarrow S. \rightarrow S. \rightarrow S$

3. Lambda-Conversion

The lambda-calculus has one very general axiom (schema), known as *lambda-conversion*, given as follows (in its monadic form).

1. Monadic Form

$\forall v \{ [\lambda v \mathcal{E}]\langle v \rangle = \mathcal{E} \}$	[(monadic) lambda-conversion axiom]
--	-------------------------------------

Here, v is a variable, and \mathcal{E} is an expression, and '=' is a generalized-identity predicate, which authorizes substitution in all contexts (in the style of Leibniz's Law).

The following is the corresponding rule, which is obtained by universal-instantiation.

$[\lambda v \mathcal{E}]\langle \Sigma \rangle = \mathcal{E}[\Sigma/v]$	[(monadic) lambda-conversion rule]
---	------------------------------------

Here, $type(\Sigma) = type(v)$, and v is free for Σ in \mathcal{E} . As usual, $\mathcal{E}[\Sigma/v]$ is, by definition, the result of substituting Σ for every occurrence of v that is free in \mathcal{E} . To say that v is free for Σ in \mathcal{E} is to say that every variable that is free in \mathcal{E} is also free in $\mathcal{E}[\Sigma/v]$. In other words, no free variable gets accidentally-bound.

2. Polyadic Form

The monadic form is a special case of the general (Polyadic) form, given as follows.

$$[\lambda v_1 \dots v_k \mathcal{E}](\langle \Sigma_1, \dots, \Sigma_k \rangle) = \mathcal{E}[\Sigma_1/v_1, \dots, \Sigma_k/v_k]$$

Here, for each i , $\text{type}(\Sigma_i) = \text{type}(v_i)$, and v_i is free for Σ_i in \mathcal{E} .

Here, $\mathcal{E}[\Sigma_1/v_1, \dots, \Sigma_k/v_k]$ results from substituting Σ_i for each occurrence of v_i in \mathcal{E} that is free for Σ_i in \mathcal{E} .

3. Examples

$$\begin{aligned} [\lambda x \mathbf{F}x](\langle a \rangle) &= \mathbf{F}a \\ [\lambda x \mathbf{R}xa](\langle a \rangle) &= \mathbf{R}aa \\ [\lambda x \mathbf{R}xx](\langle a \rangle) &= \mathbf{R}aa \\ [\lambda x \forall y \mathbf{R}xy](\langle a \rangle) &= \forall y \mathbf{R}ay \\ [\lambda P \forall x Px](\langle \lambda y \mathbf{F}y \rangle) &= \forall x [\lambda y \mathbf{F}y](\langle x \rangle) \\ &= \forall x \mathbf{F}x \\ [\lambda P \exists x Px](\langle \lambda y \mathbf{R}ya \rangle) &= \exists x [\lambda y \mathbf{R}ya](\langle x \rangle) \\ &= \exists x \mathbf{R}xa \\ [\lambda P \lambda Q \forall x \{Px \rightarrow Qx\}](\langle \lambda y \mathbf{R}ya \rangle) &= \lambda Q \forall x \{[\lambda y \mathbf{R}ya](\langle x \rangle) \rightarrow Qx\} \\ &= \lambda Q \forall x \{\mathbf{R}xa \rightarrow Qx\} \end{aligned}$$

4. Translating English into (Simple) Type-Theory

1. Note 1

Before we continue, we note that the standard language of Type-Theory, like first-order logic, is systematically impoverished. In particular:

- (1) it lacks case-markers;
- (2) it treats common-nouns as one-place predicates;
- (3) it lacks an autonomous copula-be;
- (4) it lacks an autonomous indefinite-article.

For this reason, our translations are not completely natural, as indicated in the lexicon below.

2. Note 2

Henceforth, we use bolded lower-case and small-caps letters as proper-expressions of type D [i.e., proper-nouns], and we use lower-case math-italic letters as variables of type D [i.e., individual-variables]. Also, we use bolded upper-case letters as *proper-expressions* of type $D \rightarrow S$, and we use un-bolded upper-case letters as *variables* of type $D \rightarrow S$. Finally, we use upper-case Greek letters for variables of type S.

3. Sample Lexicon

morpheme	type	translation	glossary
Jay, Kay, Elle	D	J, K, L	Jay, Kay, Elle
respects	D→(D→S)	$\lambda y \lambda x \mathbf{R}xy$	$\mathbf{R}[\alpha, \beta] =: \alpha$ respects β
admires		$\lambda y \lambda x \mathbf{A}xy$	$\mathbf{A}[\alpha, \beta] =: \alpha$ admires β
is (transitive)		$\lambda y \lambda x \{x=y\}$	
person is-a-person	C [_{df} D→S]	$\lambda x \mathbf{P}x$	$\mathbf{P}[\alpha] =: \alpha$ is a person
man is a-man		$\lambda x \mathbf{M}x$	$\mathbf{M}[\alpha] =: \alpha$ is a man
woman is-a-woman		$\lambda x \mathbf{W}x$	$\mathbf{W}[\alpha] =: \alpha$ is a woman
happy is-happy		$\lambda x \mathbf{H}x$	$\mathbf{H}[\alpha] =: \alpha$ is happy
virtuous is-virtuous		$\lambda x \mathbf{V}x$	$\mathbf{V}[\alpha] =: \alpha$ is virtuous
the-mother-of 's-mother	D→D	$\lambda x \{\mathbf{m}(x)\}$	$\mathbf{m}(\alpha) =: \text{the mother of } \alpha$
brother-of is-a-brother-of	D→(D→S)	$\lambda y \lambda x \mathbf{B}xy$	$\mathbf{B}[\alpha, \beta] =: \alpha$ is a brother of β
that, who	(D→S)→(C→C)	$\lambda P \lambda Q \lambda x (Qx \ \& \ Px)$	
[mod]	(D→S)→(C→C)	$\lambda P \lambda Q \lambda x (Qx \ \& \ Px)$	
every	C→[(D→S)→S]	$\lambda P \lambda Q \forall x (Px \rightarrow Qx)$	
some		$\lambda P \lambda Q \exists x (Px \ \& \ Qx)$	
no		$\lambda P \lambda Q \sim \exists x (Px \ \& \ Qx)$	
the	C→D	$\lambda P \iota x Px$	
not	S→S	$\lambda \Phi \sim \Phi$	
and but	S→(S→S)	$\lambda \Psi \lambda \Phi \{ \Phi \ \& \ \Psi \}$	

4. Standard Categorical-Composition

$$\begin{aligned}
 \llbracket \phi_1 + \phi_2 \rrbracket &= \llbracket \phi_1 \rrbracket \oplus \llbracket \phi_2 \rrbracket \\
 \text{where } \alpha \oplus \beta &= \llbracket \alpha \rrbracket \langle \beta \rangle && \text{if } \text{type}(\alpha) = \text{type}(\beta) \rightarrow \mathfrak{I}, \text{ for some } \mathfrak{I} \\
 &= \llbracket \beta \rrbracket \langle \alpha \rangle && \text{if } \text{type}(\beta) = \text{type}(\alpha) \rightarrow \mathfrak{I}, \text{ for some } \mathfrak{I} \\
 &= \emptyset && \text{otherwise}
 \end{aligned}$$

Here,

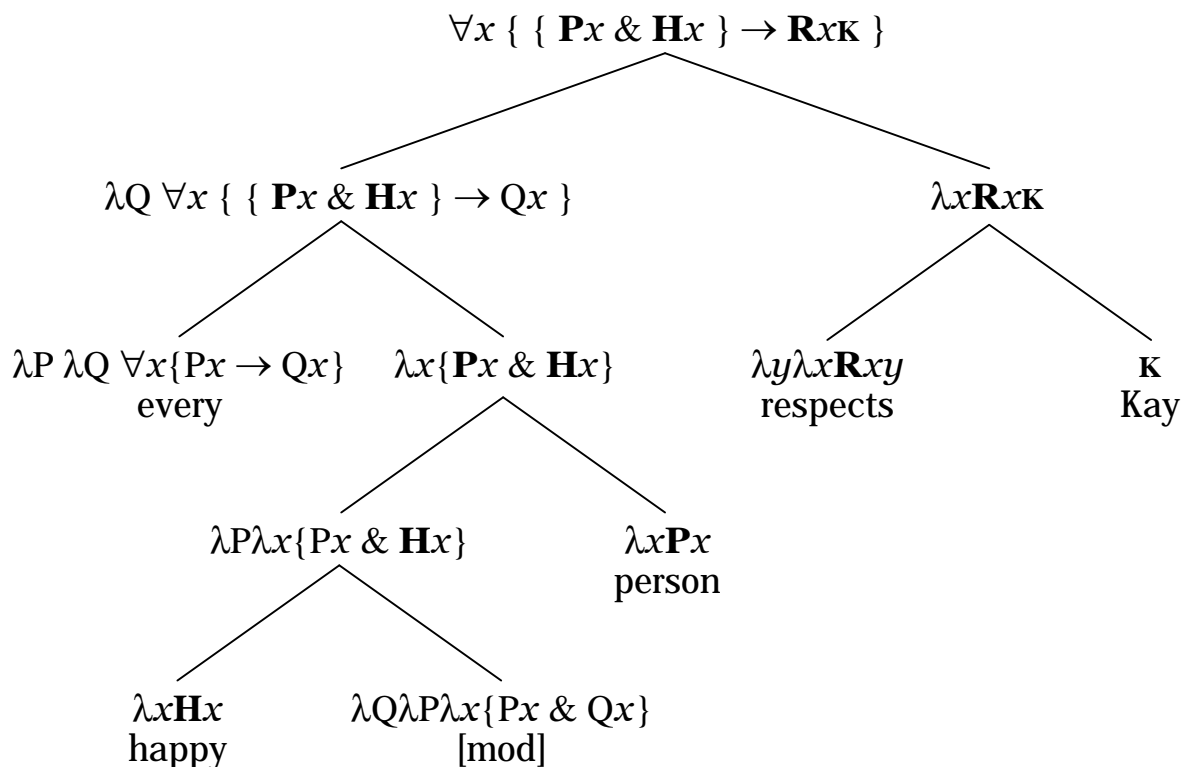
$$\begin{aligned}
 + &=_{\text{df}} \text{ syntactic composition} \\
 \oplus &=_{\text{df}} \text{ semantic composition} \\
 \llbracket \alpha \rrbracket &=_{\text{df}} \text{ the translation/meaning of } \alpha
 \end{aligned}$$

In other words, one can combine two phrases precisely if one serves as a type-appropriate argument for the other, in which case one obtains the composed meaning by applying the functor to the argument in accordance with lambda-conversion.

5. Examples

1. Example 1

every happy person respects Kay



Selected Calculations:

$$\begin{aligned}
 \llbracket \text{happy}[\text{mod}] \rrbracket &= \llbracket \text{happy} \rrbracket \oplus \llbracket \text{mod} \rrbracket \\
 &= \lambda x \mathbf{Hx} \oplus \lambda Q \lambda P \lambda x \{ Px \ \& \ Qx \} \\
 &= [\lambda Q \lambda P \lambda x \{ Px \ \& \ Qx \}] \langle \lambda x \mathbf{Hx} \rangle \\
 &= \lambda P \lambda x \{ Px \ \& \ [\lambda x \mathbf{Hx}]x \} \\
 &= \lambda P \lambda x (Px \ \& \ \mathbf{Hx})
 \end{aligned}$$

$$\begin{aligned}
 \llbracket \text{happy}_{(\text{mod})} \text{ person} \rrbracket &= \llbracket \text{happy}_{(\text{mod})} \rrbracket \oplus \llbracket \text{person} \rrbracket \\
 &= \lambda P \lambda x (Px \ \& \ \mathbf{Hx}) \oplus \lambda x \mathbf{Px} \\
 &= [\lambda P \lambda x (Px \ \& \ \mathbf{Hx})] \langle \lambda x \mathbf{Px} \rangle \\
 &= \lambda x ([\lambda x \mathbf{Px}] \langle x \rangle \ \& \ \mathbf{Hx}) \\
 &= \lambda x (\mathbf{Px} \ \& \ \mathbf{Hx})
 \end{aligned}$$

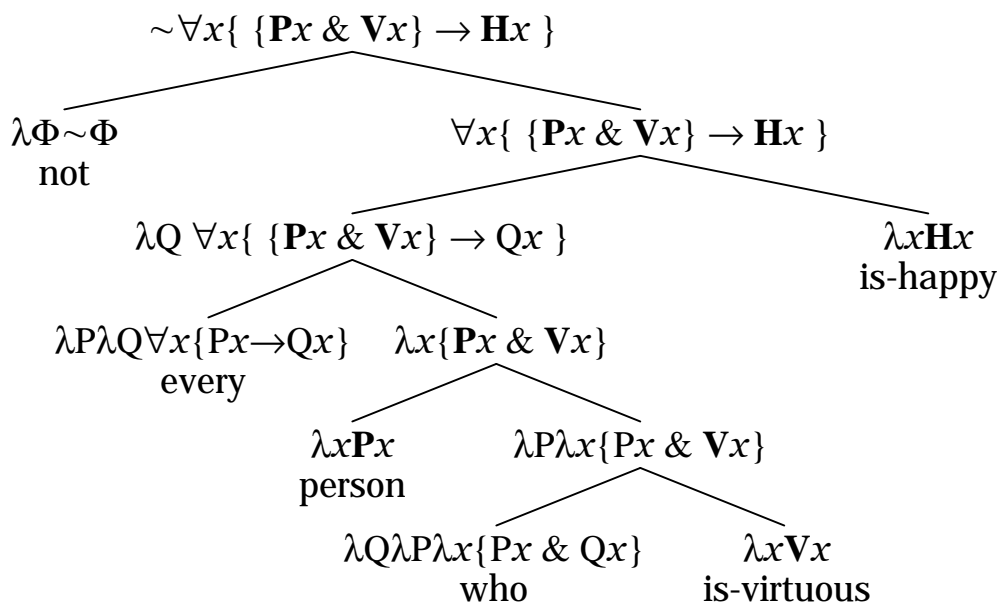
$$\begin{aligned}
 \llbracket \text{respects Kay} \rrbracket &= \llbracket \text{respects} \rrbracket \oplus \llbracket \text{Kay} \rrbracket \\
 &= \lambda y \lambda x \mathbf{Rxy} \oplus \mathbf{K} \\
 &= [\lambda y \lambda x \mathbf{Rxy}] \langle \mathbf{K} \rangle \\
 &= \lambda x \mathbf{RxK}
 \end{aligned}$$

$$\begin{aligned}
 \llbracket \text{every HP} \rrbracket &= \llbracket \text{every} \rrbracket \oplus \llbracket \text{happy person} \rrbracket \\
 &= \lambda P \lambda Q \forall x (Px \rightarrow Qx) \oplus \lambda x (\mathbf{Px} \ \& \ \mathbf{Hx}) \\
 &= [\lambda P \lambda Q \forall x (Px \rightarrow Qx)] \langle \lambda x (\mathbf{Px} \ \& \ \mathbf{Hx}) \rangle \\
 &= \lambda Q \forall x ([\lambda x (\mathbf{Px} \ \& \ \mathbf{Hx})] \langle x \rangle \rightarrow Qx) \\
 &= \lambda Q \forall x \{ (\mathbf{Px} \ \& \ \mathbf{Hx}) \rightarrow Qx \}
 \end{aligned}$$

$$\begin{aligned}
 \llbracket \text{every HP R's k} \rrbracket &= \llbracket \text{every HP} \rrbracket \oplus \llbracket \text{R's k} \rrbracket \\
 &= \lambda Q \forall x \{ (\mathbf{Px} \ \& \ \mathbf{Hx}) \rightarrow Qx \} \oplus \lambda x \mathbf{RxK} \\
 &= [\lambda Q \forall x \{ (\mathbf{Px} \ \& \ \mathbf{Hx}) \rightarrow Qx \}] \langle \lambda x \mathbf{RxK} \rangle \\
 &= \forall x \{ (\mathbf{Px} \ \& \ \mathbf{Hx}) \rightarrow [\lambda x \mathbf{RxK}] \langle x \rangle \} \\
 &= \forall x \{ (\mathbf{Px} \ \& \ \mathbf{Hx}) \rightarrow \mathbf{RxK} \}
 \end{aligned}$$

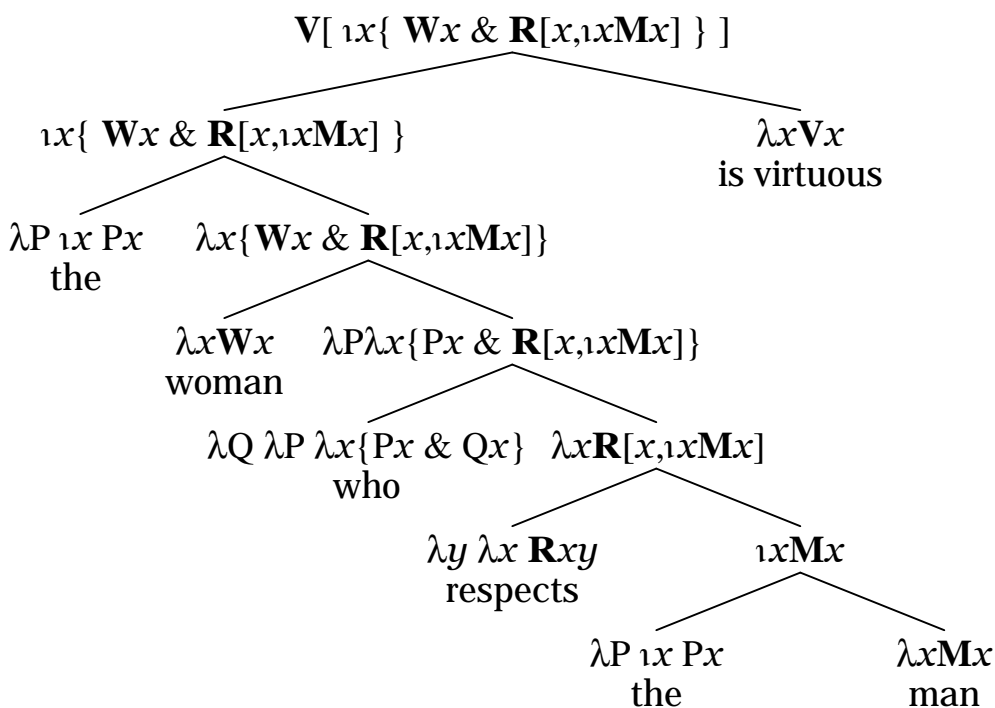
2. Example 2

not every person who is virtuous is happy



3. Example 3

the woman who respects the man is virtuous



5. Appendix – Dot-Punctuation Scheme

1. A dot serves as a parenthesis, appropriately oriented.

Whether it is a left-parenthesis or a right-parenthesis depends on which makes sense. For example:

$$\begin{array}{ll}
 A \rightarrow B \cdot \rightarrow C & \mapsto \quad A \rightarrow (B) \rightarrow C \\
 \text{and not} & \mapsto \quad A \rightarrow B (\rightarrow C) \quad \times \\
 A \rightarrow \cdot B \rightarrow C & \mapsto \quad A \rightarrow (B \rightarrow C) \\
 \text{and not} & \mapsto \quad A \rightarrow) B \rightarrow C \quad \times
 \end{array}$$

Here, ‘ \mapsto ’ is a meta-linguistic predicate that reads as “rewrites as”.

2. Parenthesis-mates are positioned accordingly.

Once a parenthesis is inserted in place of a dot, its mate is placed in the closest location that makes sense. For example:

$$\begin{array}{l} A \rightarrow B) \rightarrow C \quad \mapsto \quad (A \rightarrow B) \rightarrow C \\ \text{and not} \quad \mapsto \quad A(\rightarrow B) \rightarrow C \quad \times \\ \text{or} \quad \mapsto \quad A \rightarrow (B) \rightarrow C \quad \times \\ \text{or} \quad \mapsto \quad A \rightarrow B)(\rightarrow C \quad \times \\ \text{or} \quad \mapsto \quad A \rightarrow B) \rightarrow (C \quad \times \\ \text{or} \quad \mapsto \quad A \rightarrow B) \rightarrow C(\quad \times \end{array}$$

3. Dots may be used more than once.

For example:

$$\begin{array}{l} A \rightarrow B. \rightarrow C. \rightarrow D \\ \mapsto \\ A \rightarrow B) \rightarrow C) \rightarrow D \\ \mapsto \\ ((A \rightarrow B) \rightarrow C) \rightarrow D \end{array}$$

4. Dots can be doubled.

For example:

$$\begin{array}{l} A \rightarrow B. \rightarrow C \rightarrow D: \rightarrow D \\ \mapsto \\ A \rightarrow B) \rightarrow C \rightarrow D)) \rightarrow D \\ \mapsto \\ ((A \rightarrow B) \rightarrow (C \rightarrow D)) \rightarrow D \end{array}$$

5. Covert Dots – Minimal Dot-Punctuation Scheme

In *Minimal Form*, dots that replace left-parentheses may be elided (deleted in the overt form), whereas right-parenthesis-replacing dots are *never* elided.

$$A \rightarrow (B \rightarrow C) \quad \mapsto \quad A \rightarrow .B \rightarrow C \quad \mapsto \quad A \rightarrow B \rightarrow C$$

For example:

$$\begin{array}{l} A \rightarrow (B \rightarrow (C \rightarrow D)) \quad \mapsto \quad A \rightarrow .B \rightarrow (C \rightarrow D) \\ \mapsto \quad A \rightarrow B \rightarrow (C \rightarrow D) \\ \mapsto \quad A \rightarrow B \rightarrow .C \rightarrow D \\ \mapsto \quad A \rightarrow B \rightarrow C \rightarrow D \end{array}$$

6. Overt and covert dots can be combined.

$$\begin{array}{l} A \rightarrow B. \rightarrow C \rightarrow D \quad \mapsto \quad A \rightarrow B) \rightarrow C \rightarrow D \quad \text{first transform the overt dot} \\ \mapsto \quad (A \rightarrow B) \rightarrow C \rightarrow D \quad \text{then restore its mate} \\ \mapsto \quad (A \rightarrow B) \rightarrow .C \rightarrow D \quad \text{then restore the covert dot} \\ \mapsto \quad (A \rightarrow B) \rightarrow (C \rightarrow D) \quad \text{then transform that dot} \\ \mapsto \quad (A \rightarrow B) \rightarrow (C \rightarrow D) \quad \text{then restore its mate} \end{array}$$

7. Practical Consequence

In minimal form, dots only appear in types that are higher-order. Alternatively, dot-less conditionals are easy to read in English, dotted conditionals are hard to read. For example:

$$\begin{array}{l} A \rightarrow B \rightarrow C \quad \text{if A, then if B, then C} \\ \text{vs.} \\ A \rightarrow B. \rightarrow C \quad \text{if, if A then B, then C} \end{array}$$