

1. The Semantic Enterprise

The semantic-analysis of a phrase ϕ consists in the following.

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

2. Semantic Values – Intensions and Extensions

According to truth-conditional semantics, the *meaning (intension)* of a sentence S is identified with the conditions under which S is true/false. More generally, the meaning of a phrase ϕ is identified with the various *denotations* ϕ has under the various conditions in which ϕ might be uttered.

The notion of *denotation (extension)* is meant to consolidate and expand the notions of *reference* and *truth-value*. In particular:

- (1) The denotation of a name is the thing that bears that name.
- (2) The denotation of a sentence is its truth-value (T or F).
- (3) The denotation of a functor is a (type-appropriate) function.

For example, treating the numerals as arithmetic proper-nouns:

‘two’	denotes	the number 2
‘two plus two’	denotes	the number 4
‘two plus two is four’	denotes	the truth-value True
‘one is larger than two’	denotes	the truth-value False

3. Situations

Denotations are, by an large, **situation-dependent**. For example, whether the sentence ‘he is sitting’ is true or false depends upon **intra-loquial-factors** (including to whom the speaker is pointing) and **extra-loquial-factors** (including who is, and who is not sitting, at the time of the utterance).

A **situation** is, for us, a formal-encoding of all the utterance-dependant factors that are involved in computing the denotation of a phrase. At a minimum, a situation specifies:

- (1) the relevant **intra-loquial** information, including
 - a. the speaker ("I")
 - b. the addressee ("you")
 - c. the time ("now")
 - d. the place ("here")
 - e. demonstrations ("this", "that", "he", "she", etc.)
 - f. conversationally-salient items
 - g. the conversationally-salient universe of discourse
- (2) the relevant **extra-loquial** information

The former is often called the *context*, and the latter is often called the *possible-world*.

4. Extensional versus Intensional Semantics

- (1) In an **intensional semantics**, semantic-composition is applied to intensions.
- (2) In an **extensional semantics**, semantic-composition is applied to extensions.

We concentrate on extensional semantics first.

5. Direct versus Indirect Semantics

Another question is how do we encode and convey semantic-values? In this connection, we can proceed in two quite different ways.

- (1) indirect semantics (the method of translation)
- (2) direct semantics (the method of reference)

According to the indirect method, we **translate** each phrase of the language under consideration (the **object language**) into a corresponding phrase in a prior-understood language (the **target language**). The idea then is that the meaning of an object-phrase is the same as the meaning of the target-phrase.

Indirect semantics corresponds to how most people do semantics. If asked to give the meaning of a word or phrase, they translate or paraphrase it. For example:

- ‘bachelor’ means ‘unmarried man’ (*AHD*);¹
- ‘Apfelbaum’ in German means ‘apple tree’;
- ‘appelboom’ in Dutch means ‘apple tree’.

Indirect semantics also occurs in elementary logic, in which various English sentences are translated into logical-English.

Direct semantics is a more abstract enterprise. To do direct semantics for a given language, the object language, one uses a language, the **meta-language** – not to translate the expressions of the object-language, but rather to refer directly to the meanings/denotations of the phrases in the object language.

6. Lambda-Abstraction

Whether we pursue direct or indirect semantics, we can employ lambda-abstraction. The key difference is the syntactic-types of the lambda-expressions. In the indirect method, lambda-expressions have the same syntactic-types as the phrases they translate. In the direct method, lambda-expressions are all singular-terms that name various items of various semantic-types.

Each method has a parent theory.

- (1) type-theory for each expression in the object language, we provide a translation of that expression into a type-enriched language.
- (2) set-theory for each expression in the object language, we provide the meaning/denotation of that expression, by referring directly to that meaning using a set-theoretic expression that names it.

For the time being, we concentrate on type-theory and indirect semantics.

¹ This is short for
‘bachelor’ means what ‘unmarried man’ means.

7. Variables, Abstraction, and Order

1. Variables and Variable Subtypes

For the analysis of formal languages, we propose **variable-subtypes**.

In principle, for each type \mathfrak{I} , there is a subtype \mathfrak{I}_V of variables of that type \mathfrak{I} .

On the other hand, in first-order logic, the *only* variables are those of type D_V , which we call *individual-variables* (because they range over individuals).

2. Abstraction-Operators (Abstractors)

An **abstractor** is a functor that takes a **variable** as input and binds it.²

3. Order

$order(D)$	$=_{df}$	0
$order(S)$	$=_{df}$	0
$order(X \times Y)$	$=_{df}$	$max \{ order(X), order(Y) \}$
$order(X \rightarrow Y)$	$=_{df}$	$max \{ 1+order(X), order(Y) \}$
$order(X_V)$	$=_{df}$	$order(X)$

Examples:

$order[D \rightarrow S]$	=	1
$order[D \rightarrow (D \rightarrow S)]$	=	1
$order[D \rightarrow (D \rightarrow (D \rightarrow S))]$	=	1
$order[(D \rightarrow S) \rightarrow S]$	=	2
$order[(D \rightarrow S) \rightarrow (D \rightarrow S)]$	=	2
$order[((D \rightarrow S) \rightarrow S) \rightarrow S]$	=	3
$order[(((D \rightarrow S) \rightarrow S) \rightarrow S) \rightarrow S]$	=	4

8. Simple First-Order Abstractors

A **simple first-order abstractor** is a functor that takes an individual-variable [type D] and delivers a functor that takes a sentence/formula as input and yields a first-order expression as output, where a first-order expression has order 0 or 1.

The following are well-known examples.

abstractor	symbol	variable	input-type	output-type	type
existential quantifier	\exists	v	formula	formula	$D_V \rightarrow (S \rightarrow S)$
universal quantifier	\forall	v	formula	formula	$D_V \rightarrow (S \rightarrow S)$
definite-descriptor	ι	v	formula	singular-term	$D_V \rightarrow (S \rightarrow D)$
set-abstract ³	ζ	v	formula	singular-term	$D_V \rightarrow (S \rightarrow D)$

² The *general* theory of binding is quite complex. For current purposes, however, we merely presume the account of binding, and freedom, that is presented in every introductory symbolic logic textbook.

³ A set-abstract is usually written $\lceil \{v : \Phi\} \rceil$ or $\lceil \{v \mid \Phi\} \rceil$, but we can also write it $\lceil \zeta v \Phi \rceil$ which makes it exactly parallel to the other abstraction-operators.

9. Simple First-Order Lambda-Calculus

Simple First-Order Lambda-Calculus contains all the syntactic resources of ordinary first-order logic (FOL) plus a new variable-binding operator ‘ λ ’ (lambda), which is somewhat more complicated syntactically than the operators of FOL.

1. The Simplest Lambda-Abstracts – 1-place Predicates

1. Formation Rules

- (1) if v is a variable, and Φ is a formula, then $\lambda v\Phi$ is a one-place predicate.
- (2) if \mathbb{P} is a one-place predicate and τ is a singular-term, then $[\mathbb{P}]\langle\tau\rangle$ is a formula.

The latter is a new **general formatting-rule**, but we adopt numerous abbreviations that allow us to recover the syntactic appearance of intermediate logic.

Examples:

predicate	singular-term	formula
λxFx	a	$[\lambda xFx]\langle a \rangle$
$\lambda xRxa$	b	$[\lambda xRxa]\langle b \rangle$
$\lambda xRxx$	$f(a)$	$[\lambda xRxx]\langle f(a) \rangle$
$\lambda x\forall yRxy$	ιzPz	$[\lambda x\forall yRxy]\langle \iota zPz \rangle$

2. How to Read Predicate-Abstracts

$[\lambda v\Phi]\langle\alpha\rangle$: α is a \mathbb{N}/v such that Φ

where \mathbb{N} is a singular common-noun phrase referring to the relevant class of individuals (e.g., ‘person’, ‘number’, ‘thing’)

Examples

let:

U	=	persons
$F[\alpha]$	=	α is friendly
$R[\alpha,\beta]$	=	α respects β
a	=	Adams

predicate	reading
λxFx	___ is a person/ x such that x is friendly
$\lambda xRxa$	___ is a person/ x such that x respects Adams
$\lambda xRxx$	___ is a person/ x such that x respects x
$\lambda x\forall yRxy$	___ is a person/ x such that x respects everyone

formula	reading
$[\lambda xFx]\langle a \rangle$	Adams is a person/ x such that x is friendly
$[\lambda xRxa]\langle a \rangle$	Adams is a person/ x such that x respects Adams
$[\lambda xRxx]\langle a \rangle$	Adams is a person/ x such that x respects x
$[\lambda x\forall yRxy]\langle a \rangle$	Adams is a person/ x such that x respects everyone

2. Simple 1-place Lambda-Abstracts – Function-Signs

1. Formation Rules

- (1) if v is a variable, and Σ is a singular-term, then $\lambda v \Sigma$ is a one-place function-sign.
- (2) if \mathbb{F} is a one-place function-sign and τ is a singular-term, then $[\mathbb{F}]\langle \tau \rangle$ is a singular-term.

Examples

function-signs	singular-terms
$\lambda x \{x^2\}$	$[\lambda x \{x^2\}]\langle 2 \rangle$
$\lambda x \{x+2\}$	$[\lambda x \{x+2\}]\langle 2 \rangle$
$\lambda x \{x+x\}$	$[\lambda x \{x+x\}]\langle 2 \rangle$

2. How to Read Function-Sign-Abstracts

Although it is highly non-colloquial, the following is a decent approximation that is type-proper.⁴

$\lambda v \Sigma$ =_{df} the \mathbb{N} that is Σ supposing v is ___

Example:

$\lambda x \{x+2\}$ =_{df} the number that is $x+2$ supposing x is ___

$[\lambda x \{x+2\}]\langle 3 \rangle$ =_{df} the number that is $x+2$ supposing x is 3

3. Lambda-Conversion

$\forall v \{ [\lambda v \Phi]\langle v \rangle \leftrightarrow \Phi \}$ [lambda-conversion axioms]
 $\forall v \{ [\lambda v \Sigma]\langle v \rangle = \Sigma \}$

Examples

$\forall x \{ [\lambda x Fx]\langle x \rangle \leftrightarrow Fx \}$
 $\forall x \{ [\lambda x Rxa]\langle x \rangle \leftrightarrow Rxa \}$
 $\forall x \{ [\lambda x \forall y Rxy]\langle x \rangle \leftrightarrow \forall y Rxy \}$

$\forall x \{ [\lambda x \{x^2\}]\langle x \rangle = x^2 \}$
 $\forall x \{ [\lambda x \{x+2\}]\langle x \rangle = x+2 \}$
 $\forall x \{ [\lambda x \{x+x\}]\langle x \rangle = x+x \}$

4. Lambda-Conversion – Deduction Rule

The first form of Lambda-conversion also has a deduction-rule form, given as follows.

$[\lambda v \Phi]\langle \tau \rangle // \Phi[\tau/v]$ [lambda-conversion rule]

Here, Φ is any formula, v is any individual-variable, τ is any closed singular-term, and $\Phi[\tau/v]$ is (as usual) the formula that results when τ replaces every occurrence of v that is free in Φ .

⁴ The problem is that natural readings of $\lambda v \Sigma$ nominalize the function-sign, and are therefore type-improper.

5. Multi-Place Lambda-Abstracts

1. Formation Rules

- (1) if v_1, \dots, v_k are distinct variables, and Φ is a formula, then $\lambda v_1 \dots v_k \Phi$ is a k -place predicate.
- (2) if v_1, \dots, v_k are distinct variables, and Σ is a singular-term, then $\lambda v_1 \dots v_k \Sigma$ is a k -place function-sign.
- (3) if \mathbb{P} is a k -place predicate, and τ_1, \dots, τ_k are singular-terms, then $[\mathbb{P}]\langle \tau_1, \dots, \tau_k \rangle$ is a formula.
- (4) if \mathbb{F} is a k -place function-sign, and τ_1, \dots, τ_k are singular-terms, then $[\mathbb{F}]\langle \tau_1, \dots, \tau_k \rangle$ is a singular-term.

2. Lambda-Conversion

$$\forall v_1 \dots \forall v_k \{ [\lambda v_1 \dots v_k \Phi]\langle \tau_1, \dots, \tau_k \rangle \leftrightarrow \Phi \}$$

$$\forall v_1 \dots \forall v_k \{ [\lambda v_1 \dots v_k \Sigma]\langle \tau_1, \dots, \tau_k \rangle = \Sigma \}$$

The following is a more succinct formulation, in which \underline{v} is any sequence of distinct variables.

$$\forall \underline{v} \{ [\lambda v \Phi]\langle \underline{v} \rangle \leftrightarrow \Phi \}$$

$$\forall \underline{v} \{ [\lambda v \Sigma]\langle \underline{v} \rangle = \Sigma \}$$

Examples

$$\forall x \forall y \{ [\lambda xy \{x < y\}] \leftrightarrow x < y \}$$

$$\forall x \forall y \{ [\lambda xy \{x + y\}] = x + y \}$$

10. First-Order Lambda-Calculus

1. First-Order Abstractors

A **first-order abstractor** is a functor that takes an individual-variable [type D] and a first-order expression as input and yields a first-order output-expression, where a first-order expression has order 0 or 1.

2. Formation Rule

if v is an individual variable [type D], and Ω is a first-order expression of type \mathfrak{J} , then $\lambda v \Omega$ is an expression of type $D \rightarrow \mathfrak{J}$ (which is also first-order).

3. Examples

abstractor	variable	input	type	output	type
λ	z	$Rxyz$	S	$\lambda z Rxyz$	$D \rightarrow S$
	y	$\lambda z Rxyz$	$D \rightarrow S$	$\lambda y \lambda z Rxyz$	$D \rightarrow (D \rightarrow S)$
	x	$\lambda y \lambda z Rxyz$	$D \rightarrow (D \rightarrow S)$	$\lambda x \lambda y \lambda z Rxyz$	$D \rightarrow (D \rightarrow (D \rightarrow S))$

11. Second-Order Predicates

A second-order predicate is a functor of type $(D \rightarrow S) \rightarrow S$, which we have seen before in connection with quantifier-phrases. In order to accommodate such functors in our formal system, we must enlarge our logical machinery. At the very least, we must add predicate-variables, and we must add abstraction over predicate-variables.

The following are example lambda-abstracts in which the variable of abstraction is a predicate-variable ‘P’ and the argument is a formula.

$$\begin{aligned} &\lambda P \{ P J \} \\ &\lambda P \{ P J \ \& \ P K \} \\ &\lambda P \{ \forall x P x \} \end{aligned}$$

These do not have type-proper readings in English, although they have type-improper readings that detour through (first-order!) property-theory.

- ... is a property that Jay has
- ... is a property that Jay has and Kay has
- ... is a property that everyone has

The key, however, is that second-order lambda-predicates behave logically just like first-order lambda-predicates, which is summarized in the following lambda-conversion principles.

$\forall V \{ [\lambda V \Phi] \langle V \rangle \leftrightarrow \Phi \}$	[lambda-conversion axiom]
$[\lambda V \Phi] \langle P \rangle // \Phi [P/V]$	[lambda-conversion rule]

Here, Φ is any formula, V is any first-order predicate-variable, and P is any first-order predicate, even a complex predicate. The following is an example derivation.

- | | | |
|-----|---|----------------|
| (1) | $[\lambda P \{ P J \}] \langle \lambda x R x x \rangle$ | Pr |
| (2) | $[\lambda x R x x] \langle J \rangle$ | 1, λC |
| (3) | $R J J$ | 2, λC |

The property-theoretic translations of these inferences are as follows.

- (1) the property of respecting oneself is a property that Jay has
- (2) Jay has the property of respecting oneself
- (3) Jay respects himself

12. Full Type-Theory

Type-theory admits variables of *every* type, and accordingly admits abstraction into *every* constituent location. Indeed, every expression of the form

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

is well-formed so long as:

- v_1, \dots, v_k are distinct variables — *no matter* what their respective types, and
- \mathcal{E} is a well-formed expression — *no matter* what its type.

The exact grammatical type of $\lambda v_1 \dots v_k \mathcal{E}$ is provided by the following general rule.

if v_1, \dots, v_k respectively have types	$\mathfrak{I}_1, \dots, \mathfrak{I}_k$
and \mathcal{E} has type	\mathfrak{I}_0
then $\lambda v_1 \dots v_k \mathcal{E}$ has type	$(\mathfrak{I}_1 \times \dots \times \mathfrak{I}_k) \rightarrow \mathfrak{I}_0$

Alternatively:

$\text{type}(\lambda v_1 \dots v_k \mathcal{E})$	$=$	$[\text{type}(v_1) \times \dots \times \text{type}(v_k)] \rightarrow \text{type}(\mathcal{E})$
--	-----	--

13. Lambda-Conversion in Type Theory

Whereas first-order lambda-calculus has numerous forms of lambda-conversion, type-theory has one *very general* form, given as follows.

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

Here, \underline{v} is any sequence of distinct variables of any types, and \mathcal{E} is any expression of any type.

Note carefully that, in type-theory, ‘=’ is a generalized-identity predicate – which has a multi-type $(\mathfrak{I} \times \mathfrak{I}) \rightarrow \mathfrak{S}$, for any type \mathfrak{I} .⁵ Central to its logic is the following substitution principle.

$\forall v_1 \forall v_2 \{ v_1 = v_2 \rightarrow \forall \mathbb{P} \{ \mathbb{P} \langle v_1 \rangle = \mathbb{P} \langle v_2 \rangle \} \}$	[identity-principle]
--	----------------------

Here, v_1 and v_2 are variables of the same type \mathfrak{I}_1 , and \mathbb{P} is a variable of type $\mathfrak{I}_1 \rightarrow \mathfrak{I}_2$.

14. Lambda-Conversion Rule (Monadic)

The following is the corresponding rule, in its simplest (*monadic*) form.

$[\lambda v \mathcal{E}_1] \langle \mathcal{E}_2 \rangle$	$=$	$\mathcal{E}_1[\mathcal{E}_2/v]$	[lambda-conversion rule]
---	-----	----------------------------------	--------------------------

Here, $\text{type}(\mathcal{E}_2) = \text{type}(v)$, and v is free for \mathcal{E}_2 in \mathcal{E}_1 .

Here, $\mathcal{E}_1[\mathcal{E}_2/v]$ is by definition the result of substituting \mathcal{E}_2 in \mathcal{E}_1 for every occurrence of v that is free in \mathcal{E}_1 .

To say that v is free for \mathcal{E}_2 in \mathcal{E}_1 is to say that every variable that is free in \mathcal{E}_2 is also free in $\mathcal{E}_1[\mathcal{E}_2/v]$. In other words, no free variable gets accidentally-bound.

Here, the symbol ‘=’ indicates that the two expressions are interchangeable in all (extensional) contexts.

⁵ Note that generalized-identity coincides with ordinary identity for singular-terms (type D), and with the biconditional for formulas (type S).