Chapter 2
Basic Categorial Semantics

Chapter 2 Basic Categorial Semantics ................................................................. 1
1. Introduction ....................................................................................................... 2
2. The Semantic Enterprise .................................................................................. 2
3. Semantic-Values ............................................................................................. 2
4. Direct versus Indirect Semantics ................................................................. 5
5. The Target Language – Loglish .................................................................... 6
6. First-Order Logic ............................................................................................ 7
7. Type-Theory [Lambda-Calculus] ................................................................. 9
8. Zero-Order Lambda-Calculus ....................................................................... 9
9. First-Order Lambda-Calculus ....................................................................... 13
10. Second-Order Lambda-Calculus .................................................................. 14
11. Semantic Trees and Semantic Derivations ............................................... 16
12. Full Type-Theory [Ω-Order Lambda-Calculus] ........................................ 16
13. Translating English into Loglish – Examples ......................................... 17
A. Appendix – Summary of Basic Categorial Semantics .................................... 20
1. Official (Inductive) Definition of Types .................................................... 20
2. Common-Noun Phrases .............................................................................. 20
3. First-Order Logic ......................................................................................... 20
4. Full Type-Theory [Ω-Order Lambda-Calculus] ......................................... 22
5. Composition Rules ...................................................................................... 24
1. Introduction

Having examined basic categorial syntax, we now turn our attention to basic categorial semantics, which is intended at a minimum to recapitulate the former. On the other hand, semantics is considerably more ambitious than syntax.\(^1\) Many phrases have the same syntactic form, but they differ radically in their meaning; there is more to meaning than form. Furthermore, semantic fluency does not consist merely in assenting/dissenting to the well-formedness of phrases; rather, it involves understanding phrases.\(^2\) The task of semantics is to explain how this happens.

2. The Semantic Enterprise

The goal of semantics is, for a given language \(\mathcal{L}\), the object-language,\(^3\) to provide a semantic-analysis of every admissible phrase \(\phi\) in \(\mathcal{L}\), where the semantic-analysis of \(\phi\) consists of three inter-locking constituents.

\begin{enumerate}
  \item to provide a semantic-value for \(\phi\).
  \item to provide a semantic-value for every component-phrase of \(\phi\).
  \item to demonstrate how (1) is computed from (2).
\end{enumerate}

Note also that we presume that every phrase decomposes ultimately into fundamental (elementary) phrases, usually called morphemes,\(^4\) whose meanings are provided by the lexicon.\(^5\)

3. Semantic-Values

What are semantic-values? Fundamental to our approach is the truth-conditional model of semantics, which is summarized as follows.\(^6\)

\begin{enumerate}
  \item the fundamental units of significance are denotations.
  \item the extension of a phrase is its denotation.\(^7\)
  \item the intension of a phrase is its extension for every occasion of usage.
\end{enumerate}

We propose to expand this model by adding the following clause.

\begin{enumerate}
  \setcounter{enumi}{3}
  \item the meaning of a phrase is its intensional-tree.
\end{enumerate}

The latter is a tree each node of which is an intension.

1. Extension (Denotation)

The notion of denotation is intended to consolidate and expand the notions of reference and truth-value. In particular:

\begin{enumerate}
  \item the denotation of a name is its reference (the thing that bears that name).
  \item the denotation of a sentence is its truth-value (T or F).
\end{enumerate}

For example, treating number-words as arithmetic proper-nouns, we have the following:

---

\(^1\) This is not to suggest that semantics is theoretically harder than syntax.
\(^2\) Barbara Partee once said that Emmon Bach once said that the fundamental semantic rule is “try-to-understand”.
\(^3\) This terminology is borrowed from Logic. The object-language is the language under consideration, which contrasts with the meta-language, which is the language we employ to discuss the object-language. Most linguists use English as their meta-language, using this language to discuss hundreds of object-languages, too numerous to mention here.
\(^4\) This means that semantics spans the border between syntax and morphology. Briefly, whereas syntax analyzes phrases into words, morphology analyzes words into word-components, the smallest of which are morphemes. The division is theoretically both subtle and important. A simple example of the distinction is the difference between ‘dark room’ [syntax] and ‘darkroom’ [morphology]; note the difference in pronunciation.
\(^5\) Accordingly, the lexicon would include entries for morphemes like ‘er’, ‘ed’, ‘un’, ‘im’, ‘dis’, …
\(^6\) The truth-conditional model traces to Rudolf Carnap (1891-1970) [e.g., \textit{Meaning and Necessity: a Study in Semantics and Modal Logic}, Chicago : University of Chicago Press, 1947], whose work is inspired by Gottlob Frege (1848-1925) [e.g., “Über Sinn und Bedeutung”, \textit{Zeitschrift für Philosophie und Philosophische Kritik} 100 (1892): 25-50].
\(^7\) We take ‘denotation’ and ‘extension’ to be synonymous.
two
two plus two
two plus two is four
one is larger than two

the number 2
the number 4
the truth-value T
the truth-value F

2. Intension

The above are examples of phrases with fixed-denotations. Other such phrases include logical terms – e.g., ‘the’, ‘every’, and ‘if’. On the other hand, most phrases have variable-denotations, or occasion-dependent denotations. For example, the phrase ‘my dog’ denotes different things according to who is speaking and when. When I use the phrase ‘my dog’, it refers to my dog, insofar as I have a dog at the time of the utterance, and when you use this phrase it refers to your dog, insofar as you have a dog at the time of the utterance.

The central thesis of the truth-conditional model is that the intension of a sentence S is identified with the conditions under which S is true/false. More generally, the intension of a phrase φ is identified with the various denotations φ has under the various conditions in which φ may be uttered. We can describe this mathematically by saying that the intension of phrase φ is a function that takes each possible occasion of φ-use and yields what φ denotes on that occasion.

3. Situations

Denotations are, by an large, occasion-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 or alluding, 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-occasion-dependant factors that are involved in computing the denotation of a phrase. At a minimum, a situation specifies the following.

(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 ("factual") information

The former may be called the context (of utterance), and the latter may be called the possible-world.

8 In saying that a phrase has a fixed denotation, we don't mean it has a single denotation. Also, a given phrase may be ambiguous, in which case it has multiple denotations. Even morphemes may be lexically ambiguous in the sense that they have more than one entry in the lexicon; for example, the word ‘one’ has 24 entries at dictionary.com.

9 Another contextual factor may be whether we presuppose possession is exclusive in the sense that ‘my dog’ means ‘the unique dog that I possess’. Even possession is context-dependent. For example, if I host a dinner and ask my guests to raise their glasses for a toast, I am not asking them to raise glasses they legally own. I do not ask my guests to bring their own dinnerware!

10 Supposing the pronoun ‘he’ is demonstrative. If ‘he’ is anaphoric, then there are other considerations involved in evaluating the denotation of ‘he is sitting’, which we discuss later [Chapter 7: Pronouns].

11 Real-world situations are dynamic in character. For example, in a conversation, the words ‘I’ and ‘you’ change denotation according to who is speaking, and often the words ‘now’ and ‘here’ change denotations. Accordingly, a thorough account of situations must be dynamic in character. Our account is basically static, but it can be generalized to a dynamic account fairly easily; see Chapter ++.

12 David Kaplan (1989) distinguishes character from content. Content is a function from possible worlds to denotations, and character is a function from contexts to contents.
4. Meanings

The received view maintains that the meaning of a phrase is its intension. For example, the meaning of a sentence is just its truth-conditions. This faces a number of difficulties.

1. Logical truths all have the same truth-conditions, but they don't all have the same meaning.
2. The following sentences have the same truth-conditions.
   Kay respects herself
   Kay respects Kay
   Yet they do not have the same meaning. Phrases with the same meaning are intersubstitutable. But the above sentences are not intersubstitutable, since the following sentences are not equivalent.
   Kay respects herself, and so does Elle
   Kay respects Kay and so does Elle.

In light of these considerations, we propose to identify meanings, not with intensions, but with intensional-trees, which in particular take into account the manner of construction.

5. Composition and Compositionality

Recall that a semantic analysis of a phrase $\phi$ must:
1. provide the meaning of $\phi$,
2. provide the meanings of all the components of $\phi$,
AND
3. demonstrate how 1 is computed from 2.

The third item is critical to an account of how the mind is able to "store" an infinite amount of semantic information. It underwrites the following account.

The mind's semantic module consists in:
1. semantic-values for finitely-many simple meanings (morphemes).
2. finitely-many rules for constructing complex meanings;
AND
3. these rules are computationally-realizable.

The second component is usually described as semantic-composition, and the computational requirement is sometimes described as requiring that the semantics is compositional. This opus aims to provide a compositional semantics for a fragment of English. Note, however, that we reject the received account of compositionality.

In a compositional semantics:

<table>
<thead>
<tr>
<th>the meaning of a compound phrase</th>
</tr>
</thead>
<tbody>
<tr>
<td>is a $\times$function$\times$ of</td>
</tr>
<tr>
<td>the meanings of its immediate parts.</td>
</tr>
</tbody>
</table>

We reject this account of compositionality, in particular the term ‘function’. The usual conception of function includes the notion that a given input generates a unique output. Our conception of compositionality allows a given input to generate multiple outputs. An analogy is useful. It is well-known that classical sentential logic is computable. Whether an argument form is valid – i.e., whether a given conclusion follows from given premises – can be decided computationally. On the other hand, a given set of premises computationally-yield many,\(^13\) valid conclusions, not just one. So the output of a valid argument is not a function of the input. By analogy, although the output of semantic-composition is computationally-generated from the input, it is not a function of the input; the same input-meanings can yield multiple output-meanings.\(^14\) The key is not uniqueness-of-production, but manner-of-production.

---

\(^13\) Indeed, infinitely-many.
\(^14\) This means that semantic-composition is inherently ambiguous.
Our adjusted account of compositionality goes as follows.

<table>
<thead>
<tr>
<th>In a compositional semantics</th>
<th>the meaning of a compound phrase</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>is <strong>computationally generated</strong> from</td>
</tr>
<tr>
<td></td>
<td>the meanings of its immediate parts.</td>
</tr>
</tbody>
</table>

### 6. Extensional Semantics versus Intensional Semantics

Another key idea in semantics is the distinction between extensional semantics and intensional semantics, which may be described roughly as follows.

<table>
<thead>
<tr>
<th>extensional semantics</th>
<th>semantic-composition acts upon</th>
<th>extensions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>intensional semantics</td>
<td>semantic-composition acts upon</td>
<td>intensions.</td>
</tr>
</tbody>
</table>

The approach taken in this work is *primarily* extensional at heart. Although this approach has serious shortcomings,\(^\text{15}\) extensional semantics offers a very useful foundation for all semantic investigations.

### 4. 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:

<table>
<thead>
<tr>
<th>the phrase:</th>
<th>means:</th>
</tr>
</thead>
<tbody>
<tr>
<td>bachelor</td>
<td>unmarried man(^\text{16})</td>
</tr>
<tr>
<td>Apfelbaum</td>
<td>apple tree</td>
</tr>
<tr>
<td>appelboom</td>
<td>apple tree</td>
</tr>
<tr>
<td>XIV</td>
<td>fourteen</td>
</tr>
<tr>
<td>14</td>
<td>fourteen</td>
</tr>
</tbody>
</table>

Indirect semantics also occurs in elementary logic, in which various English sentences are translated into Logical-English ("Loglish").

Direct semantics is a considerably 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 *formally specify* the meanings of the phrases in the object language.

By way of illustrating the difference, consider the meaning of the Roman numeral ‘XIV’. According to indirect semantics, using English as the target language, we would say:

\(^{15}\) The shortcomings pertain to modal, temporal, and other functors that act on intensions rather than extensions.

\(^{16}\) It’s a bit more complicated, since some unmarried men are not *eligible* to marry, including the Pope. Is he a bachelor?
taking for granted what ‘fourteen’ means. Direct semantics goes on to analyze what ‘fourteen’ means, using mathematical English, as follows, where the first entry treats it as a proper-noun, and the second entry treats it as an adjective.\(^17\)

<table>
<thead>
<tr>
<th>fourteen</th>
<th>denotes</th>
<th>the number 14</th>
</tr>
</thead>
<tbody>
<tr>
<td>fourteen</td>
<td>denotes</td>
<td>the function that assigns True to every set that contains 14 elements, and False to every set that does not contain 14 elements</td>
</tr>
</tbody>
</table>

Our approach will follow Indirect Semantics,\(^18\) pursuing the following strategy.

<table>
<thead>
<tr>
<th>(1)</th>
<th>Object Language</th>
<th>English</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2)</td>
<td>Target Language</td>
<td>We propose a formal language, Loglish, which is gradually expounded over the course of this opus.(^19)</td>
</tr>
<tr>
<td>(3)</td>
<td>Lexicon</td>
<td>We propose translations of various morphemes into (2), which are gradually expounded over the course of this opus.</td>
</tr>
<tr>
<td>(4)</td>
<td>Composition</td>
<td>We propose a general method by which the translations/meanings of compound phrases are computationally-generated from the translations/meanings of their components, which is gradually expounded over the course of this opus.</td>
</tr>
</tbody>
</table>

5. The Target Language – Loglish

The target language we propose is called Loglish, which is a hybrid\(^20\) of English and Logic, whose precise nature is not completely known, partly because English is not a completed enterprise, but more importantly because logic is not a completed enterprise. Our aim in this work is to gradually develop an increasingly complex account of Logic, and hence Loglish.

As a first-approximation, the logic-half of Loglish is Classical First-Order Logic, as taught in elementary logic classes. This is rejected almost immediately, being replaced by a more promising second-approximation, according to which the logic-half of Loglish is Type-Theory.

As we see in the next chapter, this too proves to be inadequate. So, in subsequent chapters, we seek/offer ever improving approximations, which will involve numerous advances in logical techniques, and in semantic techniques, driven entirely by our objective – to provide an adequate target language by which to account for the semantics of English.

\(^{17}\) This is not quite the final word on number-words; see Chapter 9 [Number Words].

\(^{18}\) The (direct) semantics of our target language (Loglish) is given in a later chapter [+++].

\(^{19}\) By ‘gradually expounded’, we mean in particular that earlier proposals will often be replaced by later (better!) proposals.

\(^{20}\) One might say Loglish is a creole, but creole languages are thought to be created naturally – usually by children, presumably in accord with their innate linguistic aptitude.
6. **First-Order Logic**

As our initial approximation, we propose that the logic-half of Loglish is First-Order-Logic, which is formally described as follows.  

### 1. Underlying Categories

<table>
<thead>
<tr>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) proper-nouns</td>
</tr>
<tr>
<td>(2) variables</td>
</tr>
<tr>
<td>(3) (k)-place predicates (for each number (k \geq 0))</td>
</tr>
<tr>
<td>(4) (k)-place function-signs (for each number (k \geq 0))</td>
</tr>
<tr>
<td>(5) connectives</td>
</tr>
<tr>
<td>(6) formulas</td>
</tr>
<tr>
<td>(7) abstractors (variable-binding operators)</td>
</tr>
</tbody>
</table>

### 2. Rules of Formation (Syntax)

#### 1. Terms\(^{22}\)

1. every proper noun is a term.
2. every variable is a term.
3. if \(F\) is a \(k\)-place function-sign, and \(\tau_1, \ldots, \tau_k\) are terms, then \(F(\tau_1, \ldots, \tau_k)\) is a term.
4. if \(\Phi\) is a formula, and \(\nu\) is a variable, then \(\nu \Phi\) is a term.
5. nothing else is a term.

#### 2. Atomic Formulas

1. if \(P\) is a \(k\)-place predicate, and \(\tau_1, \ldots, \tau_k\) are terms, then \(P[\tau_1, \ldots, \tau_k]\) is an atomic formula.
2. if \(\tau_1\) and \(\tau_2\) are terms, then \([\tau_1 = \tau_2]\) is an atomic formula.
3. nothing else is a formula.

#### 3. Formulas

1. every atomic formula is a formula.
2. if \(\Phi\) is a formula, then so is: \(\neg \Phi\).
3. if \(\Phi_1\) and \(\Phi_2\) are formulas, then so are: \((\Phi_1 \& \Phi_2), (\Phi_1 \lor \Phi_2), (\Phi_1 \rightarrow \Phi_2), (\Phi_1 \leftrightarrow \Phi_2)\).
4. if \(\Phi\) is a formula, and \(\nu\) is a variable, then the following are formulas: \(\forall \nu \Phi, \exists \nu \Phi\).
5. nothing else is a formula.

---

\(^{21}\) In presenting the syntax of First-Order Logic, we do not employ categorial techniques, but rather use traditional meta-logical techniques. The reader is invited to consider how a categorial [type-governed] account might go. The main problem concerns abstractors (variable-binding operators).

\(^{22}\) Terms are often called *singular-terms*, the presumption being that First-Order Logic deals exclusively with nouns that are singular in number. Although we follow this custom in earlier chapters, we ultimately abandon this presumption, and recognize, not just singular-terms [e.g., ‘the dog’], but also plural-terms [e.g., ‘the dogs’] and mass-terms [e.g., ‘the water’]. See Chapter 9 [Number Words].
4. First-Order Languages

First-order languages are formal languages built using the syntactic rules above. They all share the following logical vocabulary in common.

(1) infinite list $v_1, v_2, \ldots$ of variables
(2) abstractor symbols: $\forall, \exists, \iota$
(3) identity sign: $=$
(4) connective symbols: $\sim, \&, \vee, \to, \leftrightarrow$
(5) punctuation symbols: $(,) [\,] , \ldots$

What distinguishes first-order languages from each other are their respective proper (non-logical) vocabularies. For any first-order language, the proper vocabulary includes.

(1.0) zero or more proper nouns
(2.0) zero or more 0-place function-signs
(2.1) zero or more 1-place function-signs
(3.0) zero or more 0-place predicates
(3.1) zero or more 1-place predicates

3. First-Order Loglish

First-order Loglish is a first-order language whose proper vocabulary consists of various lexicalized phrases of English, categorized and formatted in accord with first-order logic. The latter is aided by adopting numerous ad hoc abbreviations based on the following conventions.

<table>
<thead>
<tr>
<th>morpheme type</th>
<th>gloss</th>
</tr>
</thead>
<tbody>
<tr>
<td>D$\to$S</td>
<td>$W[\alpha]$</td>
</tr>
<tr>
<td>D$^2\to$S</td>
<td>$V[\alpha]$</td>
</tr>
<tr>
<td>D$\to$D</td>
<td>$R[\alpha,\beta]$</td>
</tr>
<tr>
<td>D$\to$D</td>
<td>$N[\alpha,\beta]$</td>
</tr>
<tr>
<td>D</td>
<td>$M(\alpha)$</td>
</tr>
<tr>
<td>D</td>
<td>$F(\alpha)$</td>
</tr>
<tr>
<td>D</td>
<td>$J$</td>
</tr>
<tr>
<td>D</td>
<td>$K$</td>
</tr>
</tbody>
</table>

The following are example morphemes.
4. **Shortcomings of First-Order Loglish**

Although First-Order Loglish can be used to translate many English phrases, it has limited syntactic resources. For example, it cannot usefully translate much of the intermediate syntactic material, including phrases like ‘respects Kay’, nor can it usefully translate logical words like ‘the’ and ‘every’.

It also involves some rather grotesque examples of syntactic *gerrymandering* – including the following proposed morphemes.

- ‘s mother
- ‘s father

7. **Type-Theory [Lambda-Calculus]**

In order to correct the shortcomings of First-Order Loglish, we enlarge the logical component of Loglish by adding the formal apparatus of Lambda-Calculus/Type-Theory, which we present incrementally in the following steps.

1. Zero-Order Lambda-Calculus
2. First-Order Lambda-Calculus
3. Second-Order Lambda-Calculus
4. Infinite-Order Lambda-Calculus

8. **Zero-Order Lambda-Calculus**

One way to expand First-Order Logic is to expand the list of variable-binding (abstraction) operators,

1. ∀ universal-quantifier operator
2. ∃ existential-quantifier operator
3. ι definite-description operator

by adding one more.

4. λ lambda-abstraction operator

The following summarizes their functional behavior.

---

27 The inability to translate ‘every’ and ‘the’ is because conventional logic treats abstractors as syn-categorimatic expressions [they don’t have types!], so the translation of English into Logic is very convoluted.

28 The term ‘calculus’ derives from the Latin word meaning ‘small stone’, and basically still means this in medicine and dentistry. Its usage in mathematics and logic derives from the ancient use of calculi (i.e., small stones) to perform calculations, and accordingly has come to mean ‘method of calculation’. The most famous such methods were co-invented by Newton and Leibniz, and were called *The Differential Calculus* and *The Integral Calculus*, and later simply *The Calculus*, and later still simply *Calculus*. A similar derivation is associated with the word ‘bible’, which originally meant ‘strip of papyrus’; after the Phoenician port Byblos, which prepared and exported such items. Thus, the root meaning of ‘bible’ is ‘book’, although it is often used to refer to a *special* book.
### The following are the corresponding rules of formation for lambda-abstraction.

1. If $\Phi$ is a formula, and $\nu$ is a variable, then $\lambda \nu \Phi$ is a lambda-predicate.

2. If $\Lambda$ is a lambda-predicate, and $\tau$ is a term, then $[\Lambda](\tau)$ is a formula.

The following are examples of this construction.

<table>
<thead>
<tr>
<th>lambda-predicate</th>
<th>term</th>
<th>formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda x F x$</td>
<td>$J$</td>
<td>$<a href="J">\lambda x F x</a>$</td>
</tr>
<tr>
<td>$\lambda x R x k$</td>
<td>$J$</td>
<td>$<a href="J">\lambda x R x k</a>$</td>
</tr>
<tr>
<td>$\lambda x R x x$</td>
<td>$J$</td>
<td>$<a href="J">\lambda x R x x</a>$</td>
</tr>
<tr>
<td>$\lambda x \forall y R x y$</td>
<td>$J$</td>
<td>$<a href="J">\lambda x \forall y R x y</a>$</td>
</tr>
</tbody>
</table>

How do we intuitively read such expressions? Although there is no adequate universal reading of ‘$\lambda$’ or ‘$\lambda x$’, there is a fairly useful reading of lambda-abstracts when they precede a formula.

<table>
<thead>
<tr>
<th>$\nu$</th>
<th>$\Phi$</th>
<th>reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\nu$</td>
<td>$\Phi$</td>
<td>is a $\nu$ such that $\Phi$</td>
</tr>
</tbody>
</table>

For example, the examples above can be read respectively as follows.

- $\lambda x F x$ is an $x$ such that $x$ is friendly
- $\lambda x R x k$ is an $x$ such that $x$ respects Kay
- $\lambda x R x x$ is an $x$ such that $x$ respects $x$
- $\lambda x \forall y R x y$ is an $x$ such that $x$ respects everyone

These lambda-predicates can then be applied to ‘$J$’, as we did above, to obtain the following readings.

- $[\lambda x F x](J)$ is an $x$ such that $x$ is friendly
- $[\lambda x R x k](J)$ is an $x$ such that $x$ respects Kay
- $[\lambda x R x x](J)$ is an $x$ such that $x$ respects $x$
- $[\lambda x \forall y R x y](J)$ is an $x$ such that $x$ respects everyone

Notice the following fairly trivial equivalences.

---

29 Notice that we adopt the usual elementary logic custom of dropping square-brackets, associated with predicates, when the arguments are all simple. Note, however, that we never drop round parentheses, associated with function-signs.
Hardegree, *Compositional Semantics*, Chapter 2: Basic Categorial Semantics

<table>
<thead>
<tr>
<th>Logical Form</th>
<th>Syntactic Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>( J ) is an ( x ) such that ( x ) is friendly ( J ) is friendly</td>
<td>( \lambda x Fx ) [( J ) is friendly]</td>
</tr>
<tr>
<td>( J ) is an ( x ) such that ( x ) respects ( K ) ( J ) respects ( K )</td>
<td>( \lambda x Rxx ) [( J ) respects ( K )]</td>
</tr>
<tr>
<td>( J ) is an ( x ) such that ( x ) respects ( x ) ( J ) respects ( x )</td>
<td>( \lambda x Rxx ) [( J ) respects ( x )]</td>
</tr>
<tr>
<td>( J ) is an ( x ) such that ( x ) respects everyone ( J ) respects everyone</td>
<td>( \lambda x \forall y R_{xy} ) [( J ) respects everyone]</td>
</tr>
</tbody>
</table>

These are all instances of a general principle of lambda-calculus, known as **Lambda-Conversion**, which is officially formulated as follows.\(^{30}\)

\[
[\lambda \nu \Phi(\tau)] \quad \overset{\text{//}}{\Rightarrow} \quad \Phi[\tau/\nu]
\]

1. \( \nu \) is any variable; \( \tau \) is any term; \( \Phi \) is any formula;
2. \( \Phi[\tau/\nu] \) results from substituting \( \tau \) for every occurrence of \( \nu \) that is free in \( \Phi \) for \( \tau \).\(^{31}\)

We next observe that lambda-expressions of this sort enable us to semantically-analyze NP-VP sentences, such as our current four examples, at least partly, as follows. Notice that each semantic analysis parallels the corresponding syntactic analysis. In place of a syntactic-type, each node consists of a phrase in Loglish of that type.

1. **Jay is friendly**

   ![Diagram for Jay is friendly]

   \( S \)
   \( D \)
   \( J \)
   \( \lambda x Fx \)
   \( \triangle \)

   \( D \rightarrow S \)
   \( \text{is friendly} \)
   \( \triangle \)

2. **Jay respects Kay**

   ![Diagram for Jay respects Kay]

   \( S \)
   \( D \)
   \( J \)
   \( \lambda x Rxx \)
   \( \triangle \)

   \( D \rightarrow S \)
   \( \text{respects Kay} \)
   \( \triangle \)

3. **Jay respects himself**

   ![Diagram for Jay respects himself]

   \( S \)
   \( D \)
   \( J \)
   \( \lambda x Rxx \)
   \( \triangle \)

   \( D \rightarrow S \)
   \( \text{respects himself} \)
   \( \triangle \)

4. **Jay respects everyone**

   ![Diagram for Jay respects everyone]

   \( S \)
   \( D \)
   \( J \)
   \( \lambda x \forall y R_{xy} \)
   \( \triangle \)

   \( D \rightarrow S \)
   \( \text{respects everyone} \)
   \( \triangle \)

---

\(^{30}\) This is a special case; see later for the general formulation. This schema is understood as a bi-directional rule [‘//’ is like identity,] licensing inter-substitution of the flanking expressions in all contexts.

\(^{31}\) See Appendix for the official account of bondage and freedom.
Notice in each example, the top node is computed from its immediate subordinate nodes (daughters) by function-application, which is formulated as follows.

<table>
<thead>
<tr>
<th>Function-Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>a phrase ( \Lambda ) of type ( A \rightarrow B )</td>
</tr>
<tr>
<td>combines with a phrase ( \alpha ) of type ( A )</td>
</tr>
<tr>
<td>to produce the phrase ( [\Lambda] \langle \alpha \rangle ) of type ( B )</td>
</tr>
</tbody>
</table>

The resulting expression is, moreover, subject to lambda-conversion. So the derivation in the last example looks thus.

<table>
<thead>
<tr>
<th>( \lambda x \forall y Rxy )</th>
<th>input phrases</th>
</tr>
</thead>
<tbody>
<tr>
<td>( [\lambda x \forall y Rxy] \langle \rangle )</td>
<td>function-application</td>
</tr>
<tr>
<td>( \forall y Rxy )</td>
<td>lambda-conversion</td>
</tr>
</tbody>
</table>

Lambda-abstracts can also be used to render simple compound-nouns such as ‘woman pilot’ and ‘gentleman farmer’, as follows.

5. woman pilot

6. gentleman farmer

Note carefully that these examples treat common-noun-phrases, not as a primitive type, \( C \), as we did in Chapter 1, but as a variant of type \( D \rightarrow S \) (predicates). The idea, which is common in logic and linguistics, is that, although they are quite different syntactically, common-noun-phrases and verb-phrases can both be semantically rendered as predicates.

Also, note that these compositions do not employ function-application, but rather Conjunction, which is formulated as follows.

<table>
<thead>
<tr>
<th>Conjunction</th>
</tr>
</thead>
<tbody>
<tr>
<td>a phrase ( \lambda \nu \Phi ) of type ( C \mid D \rightarrow S )</td>
</tr>
<tr>
<td>combines with a phrase ( \lambda \nu \Psi ) of type ( C \mid D \rightarrow S )</td>
</tr>
<tr>
<td>to produce the phrase ( \lambda \nu (\Phi &amp; \Psi) ) of type ( C \mid D \rightarrow S )</td>
</tr>
</tbody>
</table>
9. First-Order Lambda-Calculus

In the semantic-trees in the previous section, we leave unfinished the semantic-analysis of the verb-phrases.

- is friendly
- respects Kay
- respects himself
- respects everyone

How do we translate these into Loglish? For example, consider ‘respects’. We have a categorial account, according to which:

\[ \text{type(respects)} = D \rightarrow (D \rightarrow S) \]

But so far we don’t have a Loglish expression with this type.

By way of remedying this situation, we enlarge the scope of lambda-abstraction. In particular, we propose first-order lambda-calculus, which expands zero-order lambda-calculus in two ways.

1. It permits lambda-phrases to be prefixed by lambda-operators;
2. It permits terms to be prefixed by lambda-operators.

Note that item-1 is recursive, just like quantifier-operators. Just as we can have any finite string of quantifier-functors, we can have any finite string of lambda-functors. So we can have expressions like the following:

\[ \lambda x Rxyz \]
\[ \lambda y \lambda x Rxyz \]
\[ \lambda z \lambda y \lambda x Rxyz \]

What are the types of such expressions? Well, there is a very simple general principle for typing lambda-expressions, given as follows.

\[ \text{type}[\lambda \alpha \Omega] = \text{type}[\alpha] \rightarrow \text{type}[\Omega] \]

Here, \( \alpha \) and \( \Omega \) are expressions that can be combined with \( \lambda \). The following are examples.

- \( \text{type}[\lambda x Rxyz] = \text{type}[x] \rightarrow \text{type}[Rxyz] \)
- \( \text{type}[\lambda y \lambda x Rxyz] = \text{type}[y] \rightarrow \text{type}[\lambda x Rxyz] \)
- \( \text{type}[\lambda z \lambda y \lambda x Rxyz] = \text{type}[z] \rightarrow \text{type}[\lambda y \lambda x Rxyz] \)

We next note that First-Order Lambda-Calculus also allows terms (type D) to be prefixed by lambda-operators. For example, importing the plus-sign ‘+’ from arithmetic, the following is a nice polyadic example.

- \( \text{type}[\lambda y (x+y)] = \text{type}[y] \rightarrow \text{type}[x+y] \)
- \( \text{type}[\lambda x \lambda y (x+y)] = \text{type}[x] \rightarrow \text{type}[\lambda y (x+y)] \)

---

32 Note carefully that the description-operator is not recursive, since \( \nu \Phi \) is a term, not a formula.
Ordinary, non-mathematical, language has precious few polyadic function-signs, but it does have quite a few monadic function-signs, including the following, where ‘M(x)’ may be read ‘x’s mother’.

| type[λx:M(x)] | = | type[x → type[M(x)]] | = | D → D |

The two new lambda-constructions are illustrated in the following example.

7. Jay respects Kay’s mother

```
Jay respects Kay’s mother

[S]  \[\lambda x: M(x)\]  \[\lambda y: \lambda x: Rxy\]  \[\lambda x: M(x)\]  \[\lambda x: M(x)\]  \[\lambda x: M(x)\]
D→S  D  D  D
  \[\lambda x: M(x)\]  \[\lambda x: M(x)\]  \[\lambda x: M(x)\]
respects  D  D
  \[\lambda x: M(x)\]  \[\lambda x: M(x)\]  \[\lambda x: M(x)\]
Kay  D
  \[\lambda x: M(x)\]  \[\lambda x: M(x)\]  \[\lambda x: M(x)\]
Kay’s mother  D
  \[\lambda x: M(x)\]  \[\lambda x: M(x)\]  \[\lambda x: M(x)\]
```

Notice that every composition in this tree is accomplished by Function-Application along with Lambda-Conversion, as depicted in the following derivation.

| J  | λP : PJ |
| J  | λP : PJ & PK |
| J  | λP : ∀xPx |

10. Second-Order Lambda-Calculus

A second-order predicate is a functor of type (D→S)→S, which we have seen in Chapter 1 in connection with quantifier-phrases. In order to accommodate such functors in Loglish, we enlarge its logical machinery, by adding:

1. one-place predicate-variables
2. lambda-abstraction over predicate-variables.  

These new variables are symbolized by upper-case letters (not bolded). For example, in the following, the predicate-variable is ‘P’.

- \[\lambda P : PJ\]
- \[\lambda P : PJ & PK\]
- \[\lambda P : ∀xPx\]

As usual, we can compute the type of each expression. For example:

---

33 More generally, we also add multi-place predicate variables, and we add quantificational- and description-abstraction. See Appendix.

34 As opposed to proper (non-logical) predicates, which are bolded.
How do we read second-order lambda-abstracts? There are no general type-proper readings in English, but there are type-improper readings, which detour through (first-order!) property-theory.

<table>
<thead>
<tr>
<th>( \lambda P : P_j )</th>
<th>is a property that Jay has</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda P : P_j \land P_k )</td>
<td>is a property that Jay has and Kay has</td>
</tr>
<tr>
<td>( \lambda P : \forall x P_x )</td>
<td>is a property that everyone has</td>
</tr>
</tbody>
</table>

Adding predicate variables and abstraction enables us to translate quantifiers and quantifier-phrases into Loglish, as in the following example.

8. every man respects Kay

Notice that every composition in this tree is accomplished by Function-Application along with Lambda-Conversion, as depicted in the following derivation.
11. Semantic Trees and Semantic Derivations

By a semantic-tree for a phrase \( \phi \), we mean a tree consisting of the semantic-values of all the sub-phrases of \( \phi \).\(^{35}\) So far, we have mostly depicted trees in a manner similar to the way they are depicted in syntax textbooks, with the branching going down the page.\(^{36}\) But when we have drawn semantic-derivations, which are also trees, we have reversed this direction, and we have compressed the branching-lines so they are flat. The following summarizes this alternative graphical scheme.

- table-cells correspond to nodes;
- vertical lines delineate nodes on the same level;
- horizontal lines correspond to semantic transformations ("moves");
  - bold lines correspond to binary-moves (e.g., function-application);
  - non-bold lines correspond to unary-moves (e.g., lambda-conversion);
  - intermediate expressions may be omitted.
- composition proceeds down the page;
- branching proceeds up the page.

The following are examples of the new tabular graphing scheme, one syntactic-tree and one semantic-tree.

\[
\begin{array}{c|c|c}
\text{every} & \text{man} & \text{respects} \\
C & D & S \\
\vdash \quad (D \rightarrow S) & \rightarrow & S \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
\text{every} & \text{man} & \text{respects} & \text{Kay} \\
\lambda x \forall x (Mx \rightarrow RxK) & \lambda y \lambda x RxK & \lambda x RxK \\
\end{array}
\]

Henceforth, we employ the tabular graphing scheme to depict trees.

12. Full Type-Theory \([\Omega\text{-Order Lambda-Calculus}]\)

So far, we have zero-order, first-order, and second-order lambda-calculus. Full Type-theory is obtained by extending this construction ad infinitum.\(^{37}\) Although this rather imposing logical theory resides in the background of our account of semantics, and is officially presented in the Appendix, we mostly do not need more than second-order lambda-calculus.\(^{38}\)

---

\(^{35}\) We furthermore label the ultimate nodes with the original morphemes.

\(^{36}\) Noting, of course, that trees in nature branch going up into the sky!

\(^{37}\) In set theory, the first trans-finite ordinal is named \(\omega\) (omega), the upper-case variant of which is \(\Omega\). It is defined to be the set that contains 0, 1, 2, etc., and nothing else.

\(^{38}\) Note carefully that, as we see in subsequent chapters, second-order lambda-calculus proves to be inadequate, but not because of issues about order.
13. Translating English into Loglish – Examples

1. Example Lexicon

<table>
<thead>
<tr>
<th>morpheme</th>
<th>type</th>
<th>translation</th>
<th>gloss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jay, Kay, Elle</td>
<td>D</td>
<td>λy λx Rxy</td>
<td>x respects y</td>
</tr>
<tr>
<td>respects</td>
<td>D→(D→S)</td>
<td>λy λx Rx</td>
<td>x respects y</td>
</tr>
<tr>
<td>person</td>
<td>C</td>
<td>λxPx</td>
<td>x is a person</td>
</tr>
<tr>
<td>man</td>
<td>[D→S]</td>
<td>λxMx</td>
<td>x is a man</td>
</tr>
<tr>
<td>woman</td>
<td></td>
<td>λxWx</td>
<td>x is a woman</td>
</tr>
<tr>
<td>happy</td>
<td></td>
<td>λxHx</td>
<td>x is happy</td>
</tr>
<tr>
<td>virtuous</td>
<td></td>
<td>λxVx</td>
<td>x is virtuous</td>
</tr>
<tr>
<td>‘s-mother</td>
<td>D→D</td>
<td>λx:m(x)</td>
<td>the mother of x</td>
</tr>
<tr>
<td>friend-of</td>
<td>D→(D→S)</td>
<td>λy λx Bxy</td>
<td>x is a friend of y</td>
</tr>
<tr>
<td>every</td>
<td>C→[(D→S)→S]</td>
<td>λP λQ (∀x(Px → Qx))</td>
<td></td>
</tr>
<tr>
<td>some</td>
<td></td>
<td>λP λQ (∃x(Px &amp; Qx))</td>
<td></td>
</tr>
<tr>
<td>no</td>
<td></td>
<td>λP λQ (∼∃x(Px &amp; Qx))</td>
<td></td>
</tr>
<tr>
<td>the</td>
<td>C→D</td>
<td>λP λx Px</td>
<td></td>
</tr>
<tr>
<td>not</td>
<td>S→S</td>
<td>λP λx ∼P</td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>S→(S→S)</td>
<td>λP λQ (P &amp; Q) 39</td>
<td></td>
</tr>
<tr>
<td>is (identity)</td>
<td>D→(D→S)</td>
<td>λy λx [x=y]</td>
<td></td>
</tr>
<tr>
<td>is (predication)</td>
<td>∅</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>∅</td>
<td>∅</td>
<td></td>
</tr>
<tr>
<td>who</td>
<td>∅</td>
<td>∅</td>
<td></td>
</tr>
</tbody>
</table>

2. Semantic Oddities

Note that the last three entries are null (∅), which means these morphemes are semantically-empty, even though they perform critical syntactic functions.

(1) predicative-is
   
   Its syntactic-type is C→(D→S), which means ‘is’ converts a CNP into a VP, but we semantically treat C as a species of predicate [D→S], so ‘is’ takes a predicate and does nothing!

(2) a
   
   We don’t presently have semantic resources to account for number-words, including ‘a’ [≈ ‘one’], which is accordingly treated as semantically-empty. This means we also don’t have semantic resources to distinguish singular from plural, which means we only have singular at the moment. See Chapter 9 [Number Words].

(3) who
   
   Its syntactic-type is the reverse of predicative-is – (D→S)→C; it converts a VP into a CNP. Once again, since we treat C as a variant of D→S, this makes ‘who’ semantically empty.

---

39 Note carefully that we use upper-case letters as predicates of every degree (place), including 0-place. We accordingly depend upon context to disambiguate. Note also that a zero-place predicate is one that needs no subject to make a sentence; it is a sentence as it stands.
3. Example Trees

9. Jay respects Kay and Kay respects Elle

<table>
<thead>
<tr>
<th>Jay</th>
<th>respects</th>
<th>Kay</th>
<th>and</th>
<th>Kay</th>
<th>respects</th>
<th>Elle</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda y \lambda x Rx y )</td>
<td>( K )</td>
<td>( \lambda y \lambda x Rx y )</td>
<td>( K )</td>
<td>( \lambda x R x K )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \lambda x R x K )</td>
<td>( \lambda Q \lambda P { P &amp; Q } )</td>
<td>( R K L )</td>
<td>( \lambda P { P &amp; R K L } )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

10. every woman respects Kay

<table>
<thead>
<tr>
<th>every woman</th>
<th>respects</th>
<th>Kay</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda P \lambda Q \forall x { P x \rightarrow Q x } )</td>
<td>( \lambda x W x )</td>
<td>( \lambda y \lambda x R x y )</td>
</tr>
<tr>
<td>( \lambda Q \forall x { W x \rightarrow Q x } )</td>
<td>( \lambda x R x K )</td>
<td></td>
</tr>
</tbody>
</table>

\[ \forall x \{ W x \rightarrow R x K \} \]

11. every virtuous man respects Kay

<table>
<thead>
<tr>
<th>every</th>
<th>virtuous</th>
<th>man</th>
<th>respects</th>
<th>Kay</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda P \lambda Q \forall x { P x \rightarrow Q x } )</td>
<td>( \lambda x V x )</td>
<td>( \lambda x M x )</td>
<td>( \lambda x R x y )</td>
<td>( K )</td>
</tr>
<tr>
<td>( \lambda Q \forall x { (V x &amp; M x) \rightarrow Q x } )</td>
<td>( \lambda x R x K )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \forall x \{ (V x \& M x) \rightarrow R x K \} \]

12. no friend of Kay respects Jay

<table>
<thead>
<tr>
<th>no</th>
<th>friend-of Kay</th>
<th>respects</th>
<th>Jay</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda P \lambda Q \sim \exists x { P x &amp; Q x } )</td>
<td>( \lambda y \lambda x F x y )</td>
<td>( K )</td>
<td>( \lambda y \lambda x R x y )</td>
</tr>
<tr>
<td>( \lambda Q \sim \exists x { F x x &amp; Q x } )</td>
<td>( \lambda x F x k )</td>
<td>( \lambda x R x l )</td>
<td></td>
</tr>
</tbody>
</table>

\[ \sim \exists x \{ F x x \& R x l \} \]

13. the man next to Kay is Kay’s father

<table>
<thead>
<tr>
<th>the</th>
<th>man</th>
<th>next-to Kay</th>
<th>is</th>
<th>Kay’s father</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda x M x )</td>
<td>( \lambda y \lambda x N x y )</td>
<td>( K )</td>
<td>( \lambda x F (x) )</td>
<td></td>
</tr>
<tr>
<td>( \lambda P \lambda x P x )</td>
<td>( \lambda x N x k )</td>
<td>( \lambda y \lambda x[x=y] )</td>
<td>( F (k) )</td>
<td></td>
</tr>
<tr>
<td>( \lambda x (M x &amp; N x k) )</td>
<td>( \lambda x (M x &amp; N x k) )</td>
<td>( \lambda x [x=F(k)] )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \lambda x (M x \& N x k) = F (k) \]

14. every person who is virtuous is happy

<table>
<thead>
<tr>
<th>every</th>
<th>person who is virtuous</th>
<th>is happy</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda P \lambda Q \forall x { P x \rightarrow Q x } )</td>
<td>( \lambda x P x )</td>
<td>( \lambda x V x )</td>
</tr>
<tr>
<td>( \lambda Q \forall x { (P x &amp; V x) \rightarrow Q x } )</td>
<td>( \lambda x (P x &amp; V x) )</td>
<td>( \lambda x H x )</td>
</tr>
</tbody>
</table>

\[ \forall x \{ (P x \& V x) \rightarrow H x \} \]
15. not every person who respects Kay is happy

<table>
<thead>
<tr>
<th>not every person who respects Kay is happy</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda y \lambda x \cdot Rxy \ k \ \emptyset \ \lambda x Hx$</td>
</tr>
<tr>
<td>$\lambda p \lambda q \forall x {Px \to Qx}$</td>
</tr>
<tr>
<td>$\lambda x (Px \land Rxk)$</td>
</tr>
<tr>
<td>$\lambda x Hx$</td>
</tr>
<tr>
<td>$\forall x { (Px \land Rxk) \to Hx }$</td>
</tr>
</tbody>
</table>

16. every man who respects Jay’s father respects Kay’s mother

<table>
<thead>
<tr>
<th>every man who respects Jay’s father respects Kay’s mother</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda x Mx$</td>
</tr>
<tr>
<td>$\lambda x R[x,F(x)]$</td>
</tr>
<tr>
<td>$\lambda x { Mx \land R[x,F(x)] }$</td>
</tr>
<tr>
<td>$\lambda x Hx$</td>
</tr>
<tr>
<td>$\forall x { (Mx \land R[x,F(x)]) \to Hx }$</td>
</tr>
</tbody>
</table>
A. Appendix – Summary of Basic Categorial Semantics

1. Official (Inductive) Definition of Types

| (1) S is a type. | sentences |
| (2) D is a type. | definite noun phrases |
| (3) if A and B are types, then so is (A→B). | monadic functors |
| (4) nothing else is a type. |

2. Common-Noun Phrases

The type C [common-noun phrase] is treated as a derivative type, defined as follows.

\[ C = D \rightarrow S \]

3. First-Order Logic

At the core of all our semantic deliberations is First-Order Loglish, which is based on First-Order Logic, which is characterized as follows.

1. Underlying Categories

<table>
<thead>
<tr>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) proper-nouns</td>
</tr>
<tr>
<td>(2) variables</td>
</tr>
<tr>
<td>(3) (k)-place predicates (for each number (k \geq 0))</td>
</tr>
<tr>
<td>(4) (k)-place function-signs (for each number (k \geq 0))</td>
</tr>
<tr>
<td>(5) connectives</td>
</tr>
<tr>
<td>(6) formulas</td>
</tr>
<tr>
<td>(7) abstractors (variable-binding operators)</td>
</tr>
</tbody>
</table>

2. Rules of Formation

1. Terms

(1) every proper noun is a term.
(2) every variable is a term.
(3) if \(F\) is a \(k\)-place function-sign, and \(\tau_1, \ldots, \tau_k\) are terms, then \(F(\tau_1, \ldots, \tau_k)\) is a term.
(4) if \(\Phi\) is a formula, and \(\nu\) is a variable, then \(\nu \Phi\) is a term.
(5) nothing else is a term.

2. Atomic Formulas

(1) if \(P\) is a \(k\)-place predicate, and \(\tau_1, \ldots, \tau_k\) are terms, then \(P[\tau_1, \ldots, \tau_k]\) is an atomic formula.
   [We drop square-brackets and commas when the arguments are all simple.]
(2) if \(\tau_1\) and \(\tau_2\) are terms, then \([\tau_1 = \tau_2]\) is an atomic formula.
(3) nothing else is an atomic formula.

3. Formulas

(1) every atomic formula is a formula.
(2) if \(\Phi\) is a formula, then so is: \(\neg \Phi\).
(3) if \(\Phi_1\) and \(\Phi_2\) are formulas, then so are:
   \((\Phi_1 \& \Phi_2)\), \((\Phi_1 \vee \Phi_2)\), \((\Phi_1 \rightarrow \Phi_2)\), \((\Phi_1 \leftrightarrow \Phi_2)\).
(4) if $\Phi$ is a formula, and $\nu$ is a variable, then the following are formulas:

$$\forall \nu \Phi, \exists \nu \Phi.$$ 

(5) nothing else is a formula.

4. First-Order Languages

First-order languages are formal languages built on the syntactic rules above. They all share the following logical vocabulary in common.

(1) infinite list $\nu_1, \nu_2, \ldots$ of variables
(2) abstractor symbols: $\forall, \exists, 1$
(3) identity sign: $=$
(4) connective symbols: $\sim, \&, \lor, \to, \leftrightarrow$
(5) punctuation symbols: $(, ), [ , ] , \ldots$

What distinguishes first-order languages from each other are their respective proper (non-logical) vocabularies. For any first-order language, the proper vocabulary includes.

(1.0) zero or more proper nouns
(2.0) zero or more 0-place function-signs
(2.1) zero or more 1-place function-signs etc.
(3.0) zero or more 0-place predicates
(3.1) zero or more 1-place predicates etc.

5. First-Order Loglish

First-order Loglish is a first-order language that consists of various lexicalized phrases of English, categorized in accord with first-order logic, and syntactically-rendered in accord with the formatting rules of FOL. The latter is aided by adopting numerous ad hoc abbreviations based on the following conventions.

<table>
<thead>
<tr>
<th>phrase</th>
<th>gloss</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\textbf{W}[K]$</td>
<td>Kay is a woman</td>
</tr>
<tr>
<td>$\textbf{R}[J,K]$</td>
<td>Jay respects Kay</td>
</tr>
<tr>
<td>$\textbf{M}(K)$</td>
<td>Jay's mother</td>
</tr>
<tr>
<td>$\textbf{F}(\textbf{M}(K))$</td>
<td>Jay's mother's father</td>
</tr>
<tr>
<td>$\textbf{V}[	extbf{M}(K)]$</td>
<td>Jay's mother is virtuous</td>
</tr>
</tbody>
</table>

40 This is redundant; one does not need both proper nouns and zero-place function-signs; either will suffice syntactically.
41 By a lexicalized phrase, we mean a phrase that, notwithstanding its apparent complex syntactic structure, is treated by the theory as simple, and hence is stored "en masse" in the lexicon.
42 Henceforth, unless otherwise noted, ‘letter’ means ‘letter of the Roman alphabet’.
43 We propose to employ un-bold upper-case letters as predicate-variables, and bold-upper-case letters as predicate-constants.
4. Full Type-Theory [Ω-Order Lambda-Calculus]

Full Type-theory includes lambda-calculus for zero-order, first-order, second-order, *ad infinitum*.

1. Variables

Full Type-Theory has variables of *every* type, and admits lambda-abstraction over every type.

2. Lambda Abstraction

If \( \alpha \) is a variable of type \( A \), and \( \beta \) is an expression of type \( B \), then \( \lambda \alpha \beta \) is an expression of type \( A \rightarrow B \).

The use of colons is optional, being used to help parsing.

3. Lambda Application

If \( \Lambda \) is a lambda-abstract of type \( A \rightarrow B \), and \( \sigma \) is an expression of type \( A \), then \( [\Lambda] \langle \sigma \rangle \) is an expression of type \( B \).

We also adopt the following left-association rule for square-brackets.

\[
[\Lambda] \langle \alpha \rangle \langle \beta \rangle =_s [ [\Lambda] \langle \alpha \rangle ] \langle \beta \rangle
\]

4. Lambda-Conversion

\[
[\lambda v \epsilon \sigma] \leftrightarrow [\epsilon [\sigma/v]]
\]

(1) \( v \) is any variable;

(2) \( \sigma \) is any expression of the same type as \( v \);

(3) \( \epsilon [\sigma/v] \) results from substituting \( \sigma \) for every occurrence of \( v \) that is free in \( \epsilon \) for \( \sigma \).

This schema is understood as a bi-directional rule [’//’ is like identity.], licensing inter-substitution of the flanking expressions in all contexts.

5. Freedom and Bondage; Open and Closed Expressions

(1) Where \( \Sigma \) is an abstractor, \( v \) is a variable, and \( \epsilon \) is an expression, every occurrence of \( v \) in \( \Sigma v \epsilon \) is bound by \( \Sigma \).

(2) An occurrence of a variable \( v \) is free in expression \( \epsilon \) if and only if \( o \) is not bound by an abstractor.

(3) A variable \( v \) is free in \( \epsilon \) if and only if at least one occurrence of \( v \) is free.

(4) A variable \( v \) is free for \( \Sigma \) in \( \epsilon \) if and only if every variable that is free in \( \Sigma \) is also free in \( \epsilon [\Sigma/v] \).

(5) An expression \( \epsilon \) is closed if and only if no variable is free in \( \epsilon \).

(6) An expression \( \epsilon \) is open if and only if it is not closed.

6. Alphabetic Variants

In order to perform lambda-conversion, one must make free-for substitutions, which means that some expressions cannot be converted as they stand. Rather, they must be replaced by alphabetic variants. The basic idea is that expressions that differ only in regard to bound variables are equivalent. For example:

\[
\forall xFx = \forall yFy \\
\forall x\forall yRx = \forall y\forall zRyz
\]

The full technical definition is somewhat convoluted.
\(E_1\) and \(E_2\) are alphabetic variants of each other if and only if there is a permutation (1–1 function) \(\pi\) on the class \(V\) of variables, whose inverse is \(\pi^{-1}\), such that, for any variable \(v\), \(E_1\) results by substituting \(\pi(v)\) for every bound occurrence of \(v\) in \(E_2\), and \(E_2\) results by substituting \(\pi^{-1}(v)\) for every bound occurrence of \(v\) in \(E_1\).

7. Examples of Lambda-Conversion

| \([\lambda x F x](a)\) | \(F a\) |
| \([\lambda x R x a](a)\) | \(R a a\) |
| \([\lambda x R x x](a)\) | \(R a a\) |
| \([\lambda x \forall y R x y](a)\) | \(\forall y R a y\) |
| \([\lambda P \forall x P x](\lambda y F y)(x)\) | \(\forall x (\lambda y F y)(x)\) |
| \([\lambda P \exists x P x](\lambda y R y a)(x)\) | \(\exists x (\lambda y R y a)(x)\) |
| \([\lambda P \forall x P x](\lambda y R y a)(x)\) | \(\lambda Q \forall x (\lambda y R y a)(x) \rightarrow Q x\) |
| \([\lambda P \lambda Q \forall x (P x \rightarrow Q x)](\lambda y R y a)(x)\) | \(\lambda Q \forall x (\lambda y R y a)(x) \rightarrow Q x\) |

8. Variables

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

1. lower-case math-italic \(x, y, z, \ldots\) D
2. upper-case times-roman \(P, Q, R, \ldots\) D\(\rightarrow\)S, S

9. Connectives

If \(\Phi\) and \(\Psi\) are formulas, then so are the following.

\(\sim \Phi, [\Phi \& \Psi], [\Phi \lor \Psi], [\Phi \rightarrow \Psi], [\Phi \leftrightarrow \Psi]\)

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

10. Identity

If \(E_1\) and \(E_2\) are expressions of the same type, then \([E_1 = E_2]\) is a formula.

outer-brackets are usually dropped when the formula stands alone.

11. Quantification

If \(\Phi\) is a formula, and \(v\) is a variable, of any type, then \(\forall v \Phi\) and \(\exists v \Phi\) are formulas.

12. Definite-Descriptions

If \(v\) is a variable of type \(\exists\), and \(\Phi\) is a formula, then \(\exists^v \Phi\) is an expression of type \(\exists\).
5. Composition Rules

1. Function-Application

<table>
<thead>
<tr>
<th>a phrase</th>
<th>$\Lambda$</th>
<th>of type</th>
<th>$A \rightarrow B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>combines with a phrase</td>
<td>$\alpha$</td>
<td>of type</td>
<td>$A$</td>
</tr>
<tr>
<td>to produce the phrase</td>
<td>$[\Lambda \langle \alpha \rangle]$</td>
<td>of type</td>
<td>$B$</td>
</tr>
</tbody>
</table>

2. Conjunction

<table>
<thead>
<tr>
<th>a phrase</th>
<th>$\lambda v \Phi$</th>
<th>of type</th>
<th>$C \rightarrow [D \rightarrow S]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>combines with a phrase</td>
<td>$\lambda v \Psi$</td>
<td>of type</td>
<td>$C \rightarrow [D \rightarrow S]$</td>
</tr>
<tr>
<td>to produce the phrase</td>
<td>$\lambda v(\Phi &amp; \Psi)$</td>
<td>of type</td>
<td>$C \rightarrow [D \rightarrow S]$</td>
</tr>
</tbody>
</table>