1 MOTIVATION AND THEORETICAL TOOLS

1.1 Saturated and unsaturated expressions

Basic assumption of truth-conditional semantics: To know the meaning of a sentence is to know its truth-conditions, i.e., what the world would have to be like for it to be true.

- Question: How do we determine the truth conditions of clauses in a compositional way?

- Frege’s basic insight: saturated and unsaturated expressions

Statements in general (...) can be imagined to be split up into two parts; one complete in itself, and the other in need of supplementation, or unsaturated [emphasis, HF]. Thus, e.g., we split up the sentence

‘Caesar conquered Gaul’

into ‘Caesar’ and ‘conquered Gaul’. The second part is unsaturated [emphasis, HF] — it contains an empty place; only when this place is filled up with a proper name, or with an expression that replaces a proper name, does a complete sense [thought, HF] appear. Here too I give the name ‘function’ to the referent of this unsaturated part. In this case the argument is Caesar (Frege 1891: 139).

Frege viewed names and sentences as the two kinds of linguistic expressions that are “saturated”, i.e. “complete”, and all other kinds of linguistic expressions as “unsaturated”, i.e. as needing something else to combine with in order to form semantically complete, or “saturated”, expressions. Consequently, different parts of a sentence play different roles in the derivation of its meaning:

- predicative expressions (V, A, P, bare N): unsaturated, semantic functions
- nominal arguments (DP): saturated, semantic arguments

For instance, the meaning of frighten is a semantic function that requires two individuals as its input and produces as an output a “complete” description of a frightening of someone by someone (else).

(1) a. Molly frightened Leopold.
    b. *Molly frightened.

Since the transitive verb frighten cannot by itself be used to assert that someone frightened someone (else), a standard way of characterizing its meaning is as a rule or function: frightened(x, y). Generally, the meanings of transitive verbs are taken to be functions that map entities (the direct object’s meaning) to functions that map entities (the subject’s meaning) to truth values, i.e., in terms of two ‘stacked’ unary functions, each of which takes a single entity as its argument.

In this sense, predicative expressions, understood as “unsaturated expressions”, denote functions of one kind or another, and saturation consists in the application of a function to its arguments. This mode of semantic composition is achieved by Functional Application.

- Frege’s conjecture: All semantic composition is functional application.

This mode of semantic composition by functional application is the most basic operation in the derivation of sentence interpretations in a compositional way. It consists in applying the meaning of unsaturated, or function-denoting, expressions to the meaning of their saturated arguments, until all argument positions are saturated:

Functional application: The process of applying a function to an argument. We write the result of functional application with parentheses:

\[ f(x) \] read “f applied to x” or “f of x”.

A note on functions (see e.g., Heim and Kratzer 1998, p. 10). Whenever f is a function and x an element in its DOMAIN, we have: \( f(x) \) is the unique y such that \( <x, y> \in f \).

Given any x, there is only one y that can be paired with that x. A function is a set of ordered pairs in which no two ordered pairs \(<x,y>\) that have the same first component x have different second components.

Each function has a DOMAIN and a RANGE (CO-DOMAIN), which are sets:
DOMAIN the set of objects that can be inputs of a function
CO-DOMAIN = RANGE the set of objects that can, but need not, be outputs for some input.

A function takes an input argument from some specified domain and yields an output value from its co-domain (range).

Applying a function $f$ to an argument $x$ yields the value for that argument, which can be written as $f(x)$. $f(x)$ is also called the “value of the function $f$ for the argument $x$”, and we also say that “$f$ maps $x$ to $y$”.

$f(x) = y$ means the same thing as $\langle x, y \rangle \in f$.

Functions, like sets, can be defined in various ways. For instance, let $F$ be that function $f$ with domain $\{a,c,d\}$ such that $f(a) = f(c) = b$ and $f(d) = e$. The functions elements, i.e., sets of ordered pairs, can be listed or given in the form of a table:

$$F = \{\langle a,b \rangle, \langle c,b \rangle, \langle e,d \rangle\}$$

Generally, when $A$ is the domain and $B$ the range of $f$, we say that “$f$ is from $A$ ONTO $B$”, if for all $b$ in $B$ there is an $a$ in $A$ such that $f(a) = b$. I.e., all elements in $B$ (co-domain/range) are used. Such functions are referred to as ONTO or SURJECTIVE.

If $C$ is a superset of $f$‘s co-domain/range, we say that “$f$ is INTO (or TO) $C$”. For “$f$ is INTO (or TO) $C$”, we write $f : A \rightarrow C$.

An INJECTIVE function is a function that never maps distinct elements of its domain to the same element of its codomain.

Diagram: The four possible combinations of injective and surjective features


The interpretation of a sentence with a transitive verb like frighten and its two syntactic complements involves two instances of functional application (FA), where the order of saturation of arguments is determined by the syntactic structure in (3).

(2) $\llbracket$ Molly frightened Leopold $\rrbracket = \llbracket S \rrbracket$
FA: $\llbracket V \rrbracket (\llbracket DP_{obj} \rrbracket)$ the meaning of $V$ applied to the meaning of its direct object DP
FA: $\llbracket VP \rrbracket (\llbracket DP_{sub} \rrbracket)$ the meaning of $VP$ applied to the meaning of its subject DP
1.2 Lambda calculus

As has been observed above, the semantic interpretation of many sentences proceeds in a compositional fashion by functional application of unsaturated expressions (predicates are functions) to saturated arguments of appropriate kind.

The Greek letter lambda is: \( \lambda \). The \( \lambda \)-calculus provides an easy technical means of modelling the semantic composition procedure of functional application:

- The \( \lambda \)-abstraction applies the functional abstractor or set abstractor \( \lambda \) to formulas to make \( \lambda \)-predicates.
- \( \lambda \)-conversion (also \( \lambda \)-reduction) is used to saturate the argument variables bound by the lambda operator.

The lambda calculus was invented by Alonzo Church in the 1940s and 1950s (see Church 1940) to give a systematic way of representing functions, function-argument application, and the building up of complex functions which may take functions as arguments and give functions as values.

In 1957 John McCarthy designed the programming language LISP, based on the lambda calculus. Today the lambda calculus is used in two different but related ways by linguists and computer scientists: namely,

- to describe a logic of (logical) types, for programming languages and program execution,
- to simplify/unify the description of the semantics for some complex linguistic expressions.


**Lambda-abstraction rule** (informal statement):
\( \lambda \)-abstraction applies to formulas to make predicates. In its simplest implementation, \( \lambda \)-abstraction extends the first-order predicate language in a way that allows us to represent more complex common noun phrases (CNP’s) like man who loves Molly (head CNP + relative clause), adjective phrases like happy and smart, some verb phrases like drive fast.

**Rule of Thumb**: \( \lambda \)-expressions stand for the meaning of unsaturated expressions.

In first-order predicate logic, the meaning of a one-place predicate like \( \text{sleep} \) is the set of individuals who have the property of sleeping.

\[
\text{The name for this set: } [x \mid x \text{ sleep}] \quad \text{set builder notation}
\]

\[
\lambda x [x \text{ sleep}] \quad \text{lambda notation}
\]

There are two requirements for this way of naming of sets:

(i) we have a sentence, containing a variable, i.e., an open formula (a propositional function) like \( \text{sleep}(x) \) with the variable \( x \), and

(ii) we have a convention for marking which variable is the key variable for determining which set is to be indicated, in case there might be more than one in that sentence/open formula.

- In the predicate notation, the variable written to the left of the vertical line indicates this. Sometimes ‘\( \cdot \)’ is used instead of the vertical line:

  \[
  [x \mid x \text{ sleeps}] \quad \text{set builder notation}
  \]

  \[
  \lambda x [x \text{ sleep}] \quad \text{lambda notation}
  \]

- In the lambda notation, the relevant variable is bound by the functional abstractor or set abstractor \( \lambda \).

**Lambda-abstraction** (\( \lambda \)-abstraction):
If \( \phi \in \text{Form} \) and \( v \) is a variable, then \( \lambda v [\phi] \in \text{Pred-1} \).
\( \lambda v [\phi] \in M, g \) is the set \( S \) of all \( d \in D \) such that \( [\phi] M, g, d/v = 1 \).
The way to read lambda-term like the following ones is:

\[ \lambda x [\text{sleep}] \] “the property of sleeping”, or
“the property of being one of the sleeping entities (in the domain)”

\[ \lambda x [\text{smiled}] \] “the property of (actually) having smiled”, or
“the property of being one of the things that smiled (in the domain)”

\[ \lambda x [\text{happy}] \] “the property of (actually) being happy”, or
“the property of being one of the happy things”

This somewhat stilted way of putting it is due to the fact that we are dealing with the extensional case. Intensionally speaking, we can just say “the property of having smiled” and “the property of being happy” in all worlds, not just the actual one.

1.3 Sets and their characteristic functions

Following first-order predicate language, we speak of sets as the denotation of one-place predicates, Pred-1. A one-place predicate expresses a first-order property, i.e., a property of individuals.

In the lambda calculus, sets are understood as characteristic functions. This allows us to implement Frege’s strategy of taking function-argument application as the most basic operation used in semantic composition for natural language, which is also followed by Montague. This idea also neatly meshes with the assumptions of categorial grammar (Bach et al. 1987), invented by the Polish logicians (e.g., Kazimierz Ajdukiewicz 1935).

The type of one-place predicates (Pred-1) is the type \( \langle e, t \rangle \), or \( e \rightarrow t \). In order to view this type as a functional type, we should interpret a one-place predicate like sleep, not as directly denoting a set of individuals, but as denoting a characteristic function of a set of individuals.

- set of individuals who sleep: \( \{ x : x \text{ sleep} \} \)
- characteristic function of the set of individuals who sleep: \( \lambda x [x \text{ sleep}] \)

\( \lambda \)-expressions denote (characteristic) functions (of sets).
\( \lambda \)-expressions provide explicit specification of the functions they denote.

**Definition: A characteristic function of a set.**

There is a 1-to-1 correspondence between sets and their characteristic functions. The two notions are interdefinable, hence we can freely switch between “sets” and “characteristic functions”, and we can still say that one-place predicates denote sets, but we formalize this idea differently.

For any set \( S \) which is a subset of some domain \( D \) (or a ‘universe of discourse’), the characteristic function \( f_s \) of \( S \) is the following function on the domain \( D \):

- \( f_s(x) = 1 \) if \( x \in S \)
- \( f_s(x) = 0 \) if \( x \notin S \), or equivalently, since the universe is \( D \): \( f_s(x) = 0 \) if \( x \notin D - S \).

So the denotation of a one-place predicate (its semantic value) is a function from the domain \( D \) of individuals into the set \([0,1]\) of truth-values. Such a function is called a characteristic function on \( D \), which corresponds to a subset of \( D \), a characteristic function of sets of individuals in \( D \).

- The definition of a characteristic function: The characteristic function of a subset \( A \) of some set \( X \) maps elements of \( X \) to the range \([0,1]\).

Example: Suppose our domain \( D \) contains only three individuals, Stephen, Molly and Leopold. The domain \( D \) is then the set of objects:

\( D = \{ \ddot{s}, \ddot{m}, \ddot{l} \} \)

Suppose that Stephen and Molly are sleeping, but Leopold is awake. So with respect to this world (model), the one-place predicate \text{sleeps} has as its semantic value the following set:

\( \llbracket \text{sleep} \rrbracket = \{ \ddot{s}, \ddot{m} \} \)

Or we can specify the denotation (semantic value) of \text{sleeps} as a function of type \( e \rightarrow t \), i.e., a characteristic function on \( D \):

\[ \llbracket \text{sleep} \rrbracket = \begin{array}{c|c}
\ddot{s} & 1 \\
\ddot{m} & 1 \\
\ddot{l} & 0 \\
\end{array} \]

The above CHARACTERISTIC FUNCTION maps Stephen and Molly to 1, and Leopold to 0. It is the characteristic function of the set \( \{ x : x \text{ sleeps} \} \) (in a certain model), whereby
The set of objects that can be inputs (on the left of ‘→’): here, the set of three individuals.

The set of objects that can, but need not, be outputs for some input (on the right of ‘→’): here, the set of truth values {1, 0}.

The characteristic function of a set is a set-theoretic construct made from individuals, i.e., real individuals in the domain. Since a function is technically a set of ordered pairs, the above is simply a graphic representation of the following set:

\[
\langle \text{sleep} \rangle = \{ \langle \text{times}, 1 \rangle, \langle \text{times}, 2 \rangle, \langle \text{times}, 0 \rangle \}
\]

\[
\langle \text{sleep} \rangle = \text{the function } f \text{ from individuals to truth values such that: } f(x) = 1 \text{ if } x \in \{ x : x \text{ sleeps } \}, \text{ and } f(x) = 0 \text{ otherwise.}
\]

\[
\langle \text{sleep} \rangle = \lambda x [x \text{ sleep}]
\]

SUMMARY:

In first-order predicate logic, a one-place predicate expresses
- a first-order property, i.e., a property of individuals,
- the extension of which is a set of individuals,
- which is categorically-encoded by a characteristic function on the domain D.

1.4 Semantic types

Semantic types were introduced within Montague’s type theory, which allows us to implement Frege’s compositional strategy of taking function-argument application as the basic “semantic glue” for combining meanings.

Individuals and truth-values are taken to be basic, or so-called “saturated” types; they take no arguments, and therefore they can only function as arguments, not as predicates. The type of individuals is labeled e and the type of truth-values t. All other semantic types can be recursively derived from these basic types, according to the following rule (Montague, 1974).

- **Semantic types**
  - (i) Basic semantic types:
    - e (individuals)
    - t (truth-values): [0,1]
  - (ii) Functional (derived) semantic type: \(<a,b>\)
    - If a and b are semantic types, then \(<a,b>\) is a semantic type.
    - The functional type \(<a,b>\) (is equivalent to \(a \rightarrow b\)), is the type of function from a-type things to b-type things.

(iii) Nothing else is a semantic type.

- **Domains of denotations of expressions of various types**

Given a domain of entities D and the two truth values 0 and 1:

(i) \(D_e = D\) (i.e., e is the type of individuals)
In words: The domain of semantic values of type e is D.

\(D_t = \{0,1\}\) (i.e., t is the type of truth values)
In words: An expression of type t denotes an element of \{0,1\}

(ii) \(D_{<a,b>} = \{ f \mid f: D_a \rightarrow D_b \}\) (i.e. the set of all functions f from \(D_a\) to \(D_b\).)

(iii) Nothing else is a semantic type.

Ad (ii): An expression of type \(a \rightarrow b\) (or \(<a,b>\)) denotes a function from \(D_a\) to \(D_b\)
(from the domain of semantic values of type a to the domain of semantic values of type b).

This system of semantic typing implies that

(i) the only ontological categories that natural language needs are individuals and truth-values,
(ii) all meanings can be expressed in terms of these two ontological categories.
Some examples of types and function-argument application in English

• Type e → t

Since we interpret e → t expressions as denoting characteristic functions, we reinterpret the VP as denoting a function that applies to an e-type subject to give a truth value t:

\[
\begin{align*}
S &: t \\
NP &: e \quad \text{VP : <e, t>}
\end{align*}
\]

Syntactic rule: S → NP VP
Semantic rule: \[ S = \llbracket \text{VP} \rrbracket \llbracket \text{NP} \rrbracket \] or \[ S = 1 \text{ iff } \llbracket \text{NP} \rrbracket \in \llbracket \text{VP} \rrbracket \]

Note that

\[ \llbracket S \rrbracket = \llbracket \text{VP} \rrbracket \llbracket \text{NP} \rrbracket \]

gives exactly the same result as

\[ \llbracket S \rrbracket = 1 \text{ iff } \llbracket \text{NP} \rrbracket \in \llbracket \text{VP} \rrbracket . \]

If the NP denotation is a member of the set corresponding to the VP, then the characteristic function denoted by the VP will give the value 1 when applied to the entity denoted by the NP.

The type e → t is the semantic type of VP and also of unary predicates expressed by intransitive verbs (IV) like sleep, adjectives like red and common nouns like book. Semantic types play much the same role as syntactic categories do in syntax: they classify the expressions of the theory, thereby allowing us to control their interactions and state broad generalizations:

book : N
book : <e, t>  "the lexical item book is of category N"
book : <e, t>  "the expression book is of semantic type <e, t>"

Note: The English lexical item book translates into predicate logic as the one-place (unary) predicate book. In a type-theoretical language, book is treated as an expression of type <e, t>. (See more in Gamut I, Ch.4, for instance.)

We can identify the syntactic category N with the set of all lexical items with that category specification, and we can identify the semantic type <e, t> with the set of all logical expressions with the translation of the intransitive verb sleep, book (translation of the CNP book), red (translation of the adjective red) at type specification.

<table>
<thead>
<tr>
<th>syntactic category</th>
<th>semantic type</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>t (truth value): the type of truth values</td>
</tr>
<tr>
<td>NP</td>
<td>e (entity): the type of individuals</td>
</tr>
<tr>
<td>VP</td>
<td>e → t the type of functions from individuals to truth values</td>
</tr>
<tr>
<td>IV</td>
<td>e → t the type of functions from individuals to truth values</td>
</tr>
<tr>
<td>CNP</td>
<td></td>
</tr>
<tr>
<td>ADJ</td>
<td></td>
</tr>
</tbody>
</table>

• Type e → [e → t]

Generally, the goal is to have the semantic composition proceeding according to the syntactic structure. Syntactically, a transitive verb combines with its object to make a VP. A VP combines with its subject to make a S.

In order to figure out what the meaning of the transitive verb (TV) frighten should be, it is helpful to work backwards from the results we already know. We already know the types for S, NP, and VP.

\[
\begin{align*}
S &: t \\
NP &: e \quad \text{VP : <e, t>}
\end{align*}
\]

TV : <e, t>  NP : e

To figure out the type for TV, we use the fact that the function-argument application always involves the application of a function of type <a,b> to an argument of type a to give a result (a ‘value’) of type b.

\[
\begin{align*}
S &: t \\
NP &: e \quad \text{VP : <e, t>}
\end{align*}
\]

TV : <e, e, t>  NP : e

[ frighten ] is then a function that is of type e → [e → t] or <e, <e, t>>, which applies to the entity denoted by the object NP and gives us an e → t or <e, t> function as a result, a function from entities to truth values, i.e. a suitable VP meaning.

The tree below illustrates the basic compositional rule of Functional Application: namely, the application of a function to its arguments, whereby every time we apply a function to the denotation of its syntactic sister, the denotation of the sister replaces the variable in the function, and we erase (or cancel out) the corresponding lambda term to indicate saturation of the argument: lambda-conversion.
The function in (4) specifies that \textit{frighten} requires two individuals as its arguments in order to yield a sentence that can be evaluated as either true or false.

(4) \[\text{\textit{frighten} } = \lambda y \left[ \lambda x \left[ \text{\textit{frighten}(x,y)} \right] \right] \]

alternative equivalent notations:

\[\lambda y \left[ \lambda x \left[ \text{\textit{frighten}(x,y)} \right] \right]\]

subject \[\lambda y. \lambda x. \text{\textit{frighten}(x,y)}\]

direct object \[\lambda y. \lambda x. \text{\textit{frighten}(x,y)}\]

Given that the first-order predicate language interpretation of \textit{frighten} is a 2-place relation, its interpretation \textit{as a function} is as follows.

Let \[\text{\textit{I}}(\text{molly}) = \text{\textit{I}}(\text{leopold}) = \text{\textit{I}}(\text{stephen}) = \]

Or simply, having letters stand for the individuals in the domain, let \[\text{\textit{I}}(\text{molly}) = m, \text{\textit{I}}(\text{leopold}) = l, \text{\textit{I}}(\text{stephen}) = s.\]

Let \[\text{\textit{f}}(\text{\textit{molly}}) \quad \text{\textit{f}}(\text{\textit{leopold}}) \quad \text{\textit{f}}(\text{\textit{stephen}}).\]

\[\text{\textit{f}}(\text{\textit{molly}}) = \text{the characteristic function of the set of individuals who love Molly, i.e., the set } \{1\}\]

\[\text{\textit{f}}(\text{\textit{leopold}}) = \text{the characteristic function of the set of individuals who love Leopold, i.e., the empty set } \varnothing\]

Working backwards from the result for the S and the VP, we specify the characteristic function of the frightening relation, the set of all those ordered pairs of individuals such that the first member frightens the second one. So the characteristic function of \[\text{\textit{f}}(\text{\textit{frighten}})\]

in our model \(\text{\textit{M}}\) is the following binary function from pairs individuals to truth-values:

\[\begin{array}{c|c}
\langle l, m \rangle & \rightarrow 0 \\
\langle l, s \rangle & \rightarrow 0 \\
\langle m, m \rangle & \rightarrow 0 \\
\langle m, l \rangle & \rightarrow 0 \\
\langle m, s \rangle & \rightarrow 1 \\
\langle s, l \rangle & \rightarrow 0 \\
\langle s, s \rangle & \rightarrow 1 \\
\end{array}\]

Then we \textbf{Schönfinkel} this binary function right to left, since this is consistent with the principle of compositional semantics, given that a transitive verb first combines with a direct object, by function application, and returns a characteristic function of a set. This results in a unary function \textit{F} such that \[\text{\textit{F}}(\text{\textit{Y}})(\text{\textit{X}}) = 1 \text{ if and only if } \text{\textit{R}}(\text{\textit{X}}, \text{\textit{Y}}).\]

\[\begin{array}{c|c|c}
l & m & s \\
0 & 0 & 0 \\
1 & 1 & 0 \\
\end{array}\]

Schönfinkelandization (Currying): turning n-ary functions into multiple embedded unary functions.
SUMMARY:
A transitive verb combines with a direct object, by function application, and returns a characteristic function of a set. The type of functions from individuals to characteristic functions: \(<e, <e, t>>\). It is a complex function, a composition of two unary functions.

\[ \text{frighten] = \text{the function } f \text{ from individuals to characteristic functions such that:} \]
\[ f(y) = g_y, \text{ the characteristic function of } \{x : x \text{ likes } y\}. \]
\[ \text{type } < e, < e, t >> \]

\[ \text{frighten] ([ \text{ Leopold } ]) = \text{the function } f \text{ from individuals to truth values such that:} \]
\[ f(x) = 1 \text{ if } x \in \{x : x \text{ frighten Leopold}\}, \]
\[ \text{and } f(x) = 0 \text{ otherwise.} \]
\[ \text{type } < e, t >> \]

1.5 Lambda-abstraction and lambda-conversion

\[ \lambda \text{ ABSTRACTION} \]
\[ [ \ldots x \ldots ] \text{ stands for the open formula to which the lambda operator is attached:} \]
\[ \lambda x [ \ldots x \ldots ] \text{ In this formula, we abstracted over the variable } x. \]

\[ \lambda \text{ CONVERSION} \]
\[ \lambda x [ \ldots x \ldots ](\alpha) \text{ A formula of this form can be converted to a logically equivalent formula of the form} \]
\[ [ \ldots \alpha \ldots ] \text{ by replacing ALL the free occurrences of } x \text{ in the expression} \]
\[ [ \ldots x \ldots ] \text{ with } \alpha \text{ and then eliminating } \lambda x. \]
\[ \lambda x [ \ldots x \ldots ](\alpha) \]
\[ = [ \ldots \alpha \ldots ]. \]

The lambda-conversion rule is not a separate “stipulation”, but follows from the syntax and semantics of the lambda abstraction rule and the function-argument application rule below.

Example: \( x^2 + 1 \)

\[ \lambda x [x^2 + 1] \]
\[ \text{denotes the function that maps } x \text{ to } x^2 + 1 \]
\[ \equiv 5^2 + 1 \]
\[ \equiv 26 \]

\[ \lambda x \equiv \text{function-argument application} \]

Functional application as \( \lambda \)-conversion \((\lambda\)-reduction): A principle concerning the application of \( \lambda \)-expressions to arguments.

Examples:
\[ \lambda x [P(x)](z) \]
\[ \equiv P(z) \]
\[ \lambda x[x \text{ sleep}] \]
\[ \equiv \text{the characteristic function from individuals to truth values (of type } < e, t > \text{) such that for all individuals } y, \]
\[ \lambda x[x \text{ sleeps}] (y) \]
\[ \equiv [y \text{ sleep}] \]
\[ \equiv 1 \text{ iff } y \text{ sleeps, and } = 0 \text{ if } y \text{ does not sleep.} \]

\[ \lambda x[x \text{ sleeps}] [m] \]
\[ \equiv [ m \text{ sleep }] \]
\[ \equiv 1 \text{ iff Molly sleeps, and } = 0 \text{ if Molly does not sleep.} \]

In case of multiply unsaturated functions, the arguments appear in left-to-right order in the order of application, i.e. inner arguments (direct object, indirect object) before outer arguments (subject), and the \( \lambda \)-operators are deleted from left to right:

\[ \lambda y [\lambda x [R(x,y)]](u)(v) \]
\[ \equiv [\lambda x [R(x,u)](v) \]
\[ \equiv R(v,u) \]

The input, or the arguments of the function, is prefixed with lambdas, and the value, or the output of the function, follows the lambda terms.

Function(al) application can be used to interpret any syntactic structure with two branches: one branch is interpreted as a function, and the other branch is interpreted as a possible argument of the function. This is typically the case when the relationship between two syntactic elements is one of selection, such that one element denotes a
function that selects another element as an argument, the mode of semantic composition is functional application.

(5) \( f(\alpha, \beta) \) where \( f \) is the appropriate mode of composition

\[
\begin{array}{c}
\alpha \\
\beta
\end{array}
\]

The formal definition of Functional Application can be given as follows:

(6) **Functional Application (FA):**

\[
\text{If } \alpha \text{ is a branching node, } \{\beta, \gamma\} \text{ is the set of } \alpha \text{'s daughters, and } \llbracket \beta \rrbracket \text{ is a function whose domain contains } \llbracket \gamma \rrbracket, \text{ then } \llbracket \alpha \rrbracket = \llbracket \beta \rrbracket (\llbracket \gamma \rrbracket).
\]

(Heim & Kratzer, 1998)

\[
\begin{array}{c}
\alpha \\
\beta \\
\gamma
\end{array}
\]

The “domain” of the function refers to the set of all possible input values to the function.

(7) \[ A = \llbracket \alpha \rrbracket = \llbracket \beta \rrbracket (\llbracket \gamma \rrbracket) : b \]
\[ B = \llbracket \beta \rrbracket : <a,b> \quad C = \llbracket \gamma \rrbracket : a \]

2  **ILLUSTRATION: TYPES AND FUNCTIONAL APPLICATION**

2.1  **TOY ENGLISH**

2.1.1  **Syntax**

We specify the syntax of English as follows:

- **Labels** for syntactic categories:
  - S, NP, VP, V, Conj, Mod (= Modifier)

- **Basic expressions**:
  - Leopold, Stephen, Molly, sleeps, snores, loves, knows, and, or, it-is-not-the-case-that

- **Phrase structure rules**:
  a. \( S \rightarrow \text{NP } \text{VP} \)
  b. \( \text{VP } \rightarrow \text{V } \text{NP} \)
  c. \( S \rightarrow \text{S Conj S} \)
  d. \( S \rightarrow \text{Mod } S \)
  e. \( \text{NP } \rightarrow \{\text{Leopold, Stephen, Molly}\} \)
  f. \( \text{VP } \rightarrow \{\text{sleeps, snores}\} \)
  g. \( \text{V } \rightarrow \{\text{loves, knows}\} \)
  h. \( \text{Conj } \rightarrow \{\text{and, or}\} \)
  i. \( \text{Mod } \rightarrow \text{ it-is-not-the-case-that}\)

- **Starting symbol**: S.

We can derive well-formed expressions of Toy English by starting with the symbol S and replacing it step by step, following the phrase-structure rules, until we arrive at a string of basic expressions. Example:

(1) \[ S \quad \text{(Starting Symbol)} \]
\[ S \text{ Conj S} \quad \text{(Rule c)} \]
\[ S \text{ Conj NP VP} \quad \text{(Rule a)} \]
\[ S \text{ Conj Molly VP} \quad \text{(Rule e)} \]
\[ NP \text{ VP Conj NP VP} \quad \text{(Rule b)} \]
\[ NP \text{ VP Conj Molly VP} \quad \text{(Rule e)} \]
\[ Leopold VP Conj NP V NP \quad \text{(Rule f)} \]
\[ Leopold snores Conj NP V NP \quad \text{(Rule c)} \]
\[ Leopold snores and NP V NP \quad \text{(Rule e)} \]
\[ Leopold snores and Molly VP \quad \text{(Rule e)} \]
\[ Leopold snores and Molly V NP \quad \text{(Rule e)} \]
\[ Leopold snores and Molly loves NP \quad \text{(Rule e)} \]
\[ Leopold snores and Molly loves Stephen \quad \text{(Rule e)} \]

The resulting string, *Leopold snores and Molly loves Stephen*, is a syntactically well-formed expression of Toy English. It consists of basic expressions, and it was generated from the starting symbol S by recursive applications of the phrase-structure rules.

We applied the rules to derive our example in a particular order. But nothing hinges on the particular order that was followed. We could as well have proceeded in the following way:

(2) \[ S \quad \text{(Starting Symbol)} \]
\[ S \text{ Conj S} \quad \text{(Rule c)} \]
\[ S \text{ Conj NP VP} \quad \text{(Rule a)} \]
\[ S \text{ Conj Molly VP} \quad \text{(Rule e)} \]
\[ NP \text{ VP Conj NP VP} \quad \text{(Rule a)} \]
\[ NP \text{ VP Conj Molly VP} \quad \text{(Rule a)} \]
\[ Leopold VP and Molly VP \quad \text{(Rule c)} \]
\[ Leopold VP and Molly V NP \quad \text{(Rule c)} \]
\[ Leopold VP and Molly V Stephen \quad \text{(Rule e)} \]
\[ Leopold snores and Molly V Stephen \quad \text{(Rule f)} \]
\[ Leopold snores and Molly loves Stephen \quad \text{(Rule g)} \]
The syntax of Toy English will generate infinitely many sentences. This is due to rule (c), a recursive rule. Recursion in linguistics: phrases are embedded within phrases of the same type in a hierarchical structure.

(3)  
\[ S \rightarrow S \text{ Con}j S \text{ (recursive rule).} \]  
\[ S \text{ Con}j S \]  
\[ S \text{ Con}j S \text{ Con}j S \]  
\[ S \text{ Con}j S \text{ Con}j S \text{ Con}j S \]  
...  

The above derivations of well-formed expressions of Toy English include the order in which the phrase structure rules are applied. We can specify a derivation of a string with a phrase structure grammar by a tree that is order neutral:

(4)  
\[
\begin{array}{c}
\text{S} \\
\text{Conj} \\
\text{NP} \quad \text{VP} \\
\text{NP} \quad \text{VP} \\
\text{V} \quad \text{NP} \\
\text{Leopold} \quad \text{snores and} \quad \text{Molly} \quad \text{loves} \quad \text{Stephen}
\end{array}
\]

The phrase structure grammar of a language should lead to phrase structure trees that show syntactic groupings or constituents that are independently motivated by syntactic tests. For example, we analyze a sentence with a transitive verb like Molly loves Stephen in a way where loves and Stephen form a phrase, a VP, and not in a way where Molly and loves form a phrase. For example, there are pronominal forms like do so for a VP, but not for a purported constituent that consists of a subject NP and a transitive verb:

Molly loves Stephen, and so does Leopold.  
(i.e., Leopold loves Stephen as well; it cannot mean that Molly loves Leopold, as well!)

Trees are also specified in a linear way, with the help of brackets and indices. The above tree could have been given in the following way as well:

(5)  
\[
\begin{array}{c}
[ S [ S [ NP \text{Leopold} ] [ VP \text{snores} ] ] [ Conj ] ] [ S [ NP \text{Molly} ] [ VP [ V \text{loves} ] [ NP \text{Stephen} ] ] ]
\end{array}
\]

2.1.2 Semantic rules

RULE 1:  
\[
\begin{align*}
[ \text{Leopold} ] &= l \\
[ \text{Molly} ] &= m \\
[ \text{Stephen} ] &= s
\end{align*}
\]

- proper names are of type e, the type of individuals.

RULE 2:  
\[
\begin{align*}
[ \text{snores} ] &= \lambda x [ x \text{snores} ] \\
[ \text{snores} ] &= \lambda x [ x \text{snores} ]
\end{align*}
\]

- a type for intransitive verbs, a characteristic function, a lambda term

RULE 3:  
\[
\begin{align*}
[ [ S [ NP \alpha ] [ VP \beta ] ] ] &= [ [ VP \beta ] ] ([ [ NP \alpha ] ])
\end{align*}
\]

The semantic rule that corresponds to the syntactic rule \( S \rightarrow NP \ VP \).

RULE 4:  
If \( \alpha \) is a word and \( X \) is a syntactic label, then  
\[
[ [ X \alpha ] ] = [ \alpha ]
\]

\( X \) e.g., NP  
\( \alpha \) Leopold

The semantic rule that corresponds to the terminal node syntactic rule (a terminal node has no ‘child’ node).

RULE 5:  
\[
[ \text{loves} ] = \lambda y [ \lambda x [ x \text{loves} y ] ]
\]

RULE 6:  
\[
\begin{align*}
[ [ ADP [ V \alpha ] [ NP \beta ] ] ] &= [ [ V \alpha ] ] ([ [ NP \beta ] ])
\end{align*}
\]

The semantic rule that corresponds to the syntactic rule \( VP \rightarrow V \ NP \).

Syntactically, a transitive verb combines with its object to make a VP, and we want the semantic composition to proceed according to the syntactic structure.
2.1.3 Example of derivations

Let us now illustrate how the meaning of a complex sentence can be derived.

Let us assume that Leopold snores. Then we can compute the meaning of the syntactic tree $[S [NP Leopold] [VP snores]]$ as follows:

$[S [NP Leopold] [VP snores] ]$

$\equiv [VP snores] ([NP Leopold])$

$\equiv \lambda x [snores x](1)$

$\equiv [1 snores ]$

$\equiv 1$

(Assumption about reality)

(6)

(7) Leopold loves Molly

Assume that Leopold loves Molly is true (assumption about reality)

$[S [NP Leopold] [VP loves Molly] ]$

$\equiv [VP loves Molly] ([NP Leopold])$

$\equiv \lambda y [loves y](m)(1)$

$\equiv \lambda x [loves x](m)(1)$

$\equiv [1 loves m ]$

$\equiv 1$

(1)

(Assumption about reality)

Note that $\lambda y [loves y]$ is an unsaturated meaning (a complex function, a composition of two unary, or one-place, functions) that, when combined with an individual m, yields another unsaturated meaning (unary, or one-place, function), $\lambda x [loves m]$. Only when $\lambda x [loves m]$ is combined with an individual do we get a saturated meaning.$\lambda y [loves y]$

$\lambda y [loves y](m)$

$\equiv \lambda x [loves x](m)(1)$

the characteristic function of the set of persons who love Molly

$\equiv \lambda x [loves x](m)(1)$

true if Leopold indeed loves Molly, else false

2.2 More examples

For all the examples below (examples adapted from Partee MGU 2005), assume the model in which we have $[loves]^{M,g}$ be a set of ordered pairs: $\{<l,m>, <m, s>, <s, s>\}$, and assume that we start with an assignment $g$ such that $g(x) = 1$ for all $x$. In most of the examples below, the choice of initial assignment makes no difference.

$\lambda x [loves x](m)(1)$

$\equiv \lambda x [loves x](m)(1)$

the set of all individuals that love

$\lambda x [loves x](m)(1)$

$\equiv \lambda x [loves x](m)(1)$

the set of all individuals that love

$\lambda x [loves x](m)(1)$

$\equiv \lambda x [loves x](m)(1)$

the set of all who snore and love

Note on the use of lambda expressions: The lambda calculus is used to calculate what semantic values should be assigned to some syntactic constituent of a larger expression whose semantic value is known. It is the semantic values of the lambda expressions, rather than the lambda expressions themselves, that are identified with the semantic values of English expressions (Dowty, Wall and Peters 1981:110).
representation of *snores and sleeps*: \( \lambda y[\text{snore}(y) \land \text{sleep}(y)] \)

representation of *Molly snores and sleeps*: \( \lambda y[\text{snore}(y) \land \text{sleep}(y)](m) \)

Syntactic structure of the formula in (13):

\[
\lambda y \left[ \text{snore}(y) \land \text{sleep}(y) \right](m)
\]

In words: “\( m \) has the property of (actually) snoring and sleeping”

representation of the CNP man who loves Molly:

Syntactic structure:

\[
\lambda y \left[ \text{CNP'}(y) \land \text{REL'}(y) \right](\text{CNP'}, \text{REL'}: \text{combining “translations”})
\]

Rule for combining CNP and REL:

\[
\lambda y[\text{CNP'}(y) \land \text{REL'}(y)](\text{CNP'}, \text{REL'}: \text{combining “translations”})
\]

Compositional translation of the syntactic structure above into \( \lambda \)-calculus:

(14) representation of the CNP man who loves Molly:

\[
\lambda y[\text{CNP'}(y) \land \text{REL'}(y)](\text{CNP'}, \text{REL'}: \text{combining “translations”})
\]

\[
\text{man} \; \text{love} \left( z, m \right)
\]

By \( \lambda \)-conversion, we get:

\[
\lambda y[\text{man}(y) \land \lambda z[\text{love}(z, m)](y)] \equiv \lambda y[\text{man}(y) \land \text{love}(y, m)]
\]

Syntactic categories and their semantic types

<table>
<thead>
<tr>
<th>Syntactic category</th>
<th>Semantic type (extensionalized)</th>
<th>Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProperN</td>
<td>e</td>
<td>names (John)</td>
</tr>
<tr>
<td>S</td>
<td>t</td>
<td>sentences</td>
</tr>
<tr>
<td>CN(P)</td>
<td>( e \rightarrow t )</td>
<td>common noun phrases (cat)</td>
</tr>
<tr>
<td>NP</td>
<td>(i) e ( \rightarrow t )</td>
<td>“e-type” or “referential” NPs (John, the king)</td>
</tr>
<tr>
<td></td>
<td>(ii) ( e \rightarrow t ) ( \rightarrow t )</td>
<td>noun phrases as generalized quantifiers (every man, the king, a man, John)</td>
</tr>
<tr>
<td>ADJ(P)</td>
<td>(i) ( e \rightarrow t )</td>
<td>NPs as predicates (a man, the king)</td>
</tr>
<tr>
<td></td>
<td>(ii) ( e \rightarrow t ) ( \rightarrow t )</td>
<td>predicative adjectives (carnivorous, happy)</td>
</tr>
<tr>
<td>REL</td>
<td>( e \rightarrow t )</td>
<td>adjectives as predicate modifiers (skillful)</td>
</tr>
<tr>
<td>VP, IV</td>
<td>( e \rightarrow t )</td>
<td>relative clauses (who(m) Mary loves)</td>
</tr>
<tr>
<td>TV(P)</td>
<td>type(NP) ( \rightarrow t )</td>
<td>verb phrases, intransitive verbs (loves Mary, is tall, walks)</td>
</tr>
<tr>
<td>is</td>
<td>( e \rightarrow t ) ( \rightarrow t )</td>
<td>is</td>
</tr>
<tr>
<td>DET</td>
<td>type(CN) ( \rightarrow t )</td>
<td>a, some, the, every, no</td>
</tr>
</tbody>
</table>

References


Acknowledgements
Partee, H. Barbara. 2007b. Lecture Notes MGU.
Potts, Chris. 2007. LSA Logic Class.