1 Characteristic functions

The characteristic function of a set:
(1) Let $A$ be a set. The, $\text{char}_A$, the characteristic function of $A$, is the function $F$ such that, for any $x \in A$, $F(x)=1$, and for any $x \notin A$, $F(x)=0$.

- Any unary predicate can be treated as having a set-meaning
  - e.g. “red” denotes a set of red things
- or as having a function-meaning
  - e.g. “red” denotes a function that, for each thing, returns 1 if the thing is in the red set, and 0 if the thing is not in the red set

Schönfinkelization (Currying)
(2) $U = \{a,b,c\}$
(3) The relation “fond of”: $R_{\text{fond-of}} = \{<a,b>, <b,c>, <c,c>\}$
(4) The characteristic function of $R_{\text{fond-of}}$ (takes all combinations of two elements, gives 0/1):
   - $<a,a> \rightarrow \emptyset$
   - $<a,b> \rightarrow 1$
   - $<a,c> \rightarrow \emptyset$
   - $<b,a> \rightarrow \emptyset$
   - $<b,b> \rightarrow \emptyset$
   - $<b,c> \rightarrow 1$
   - $<c,a> \rightarrow \emptyset$
   - $<c,b> \rightarrow \emptyset$
   - $<c,c> \rightarrow 1$

Recall:
- We are treating binary, ternary etc. predicates as combining with both of their arguments at once.
  - meaning-wise, this means we can treat them as relations (sets of ordered tuples)
  - or, alternatively, as characteristic functions of relations
  - e.g., “fond-of” is a set of ordered pairs where the first element is fond of the second element
- If we want to, instead, to “feed” the arguments to the predicate one at a time, we need to pick which one we’re going to do first, and which second.
e.g., “fond-of” can be thought of as a function that combines with a person who is potentially fond of someone, then combines with the potentially-liked person, and then gives a 0 or 1, depending on whether the first person is fond of the second.

(5) Turning n-ary functions into multiple embedded 1-ary functions: Schönfinkelization.

\[
\begin{align*}
    a & \rightarrow 0 & \text{left-to-right schönfinkelization} \\
    a \rightarrow b & \rightarrow 1 \\
    c & \rightarrow 0 \\
    a & \rightarrow 0 \\
    b \rightarrow b & \rightarrow 0 \\
    c & \rightarrow 1 \\
    a & \rightarrow 0 \\
    c \rightarrow b & \rightarrow 0 \\
    c & \rightarrow 1
\end{align*}
\]

EXERCISE 1: Given the following Universe, spell out the characteristic function of the set in (6) and schönfinkelize it left-to-right.

(6) \( U = \{d, e\} \)

(7) \( R = \{<d,d,d>, <d,e,d>, <e,d,d>, <e,e,e>, <e,e,d>\} \)

4 Lambdas and types

4.1 Motivation: Set theory and thus, the semantics of Predicate Logic has problems:

Problem 1: Russel’s paradox

- There are several ways to define a set:
  - list all its members \( \{2,4,6,8\} \)
  - define via a property \( \{x \mid x \text{ is an even positive number less than 10}\} \)
- This last version is very convenient (powerful):
  - \( \{x \mid x \text{ is in this room now}\} \)
  - \( \{x \mid x \text{ is red in w100}\} \)
  - \( \{x \mid <x,m> \in \text{[[kiss]]}_w^{17}\} \)

But it can lead to problems, e.g. \( A = \{x \mid x \notin x\} \)

QUESTION 3: is \( A \in A \)?

Problem 2: Verbs, prepositions, etc. not corresponding to the same sort of semantic thing

What is the nature of syntactic categories, like verb, noun?
• We cannot say “V always combines with one DP to make S”. Instead, we must define further categories:
  ◦ \( V_{\text{intransitive}} \) combines with one DP to make S,
  ◦ \( V_{\text{transitive}} \) combines with two DPs to make S
  ◦ \( V_{\text{ditransitive}} \) combines with three DPs to make S
• Similarly, count and mass nouns behave differently:
  ◦ \( N_{\text{mass}} \) is an NP that can be a DP by itself
  ◦ \( N_{\text{count}} \) combines with Det to make DP
• Yet, there are syntactic reasons to call all verbs, “verbs” and all nouns, “nouns”
  ◦ All phrases built around a verb are VPs, all phrases built around nouns are NPs etc.
  ◦ All verbs inflect for the same things in the same language: e.g. loves, runs, gives etc.

Problem 3: too many semantic rules, or not enough!
• Combining a subject DP with the VP:
  \[
  \forall S, V, D, M \in M \\
  [S]_M = 1 \iff [D]_M \in [V]_M \\
  / \ \ / \ \ \\
  DP \ V P \\
  \]

QUESTION 4: Take the denotation of kiss in M1 to be the set of pairs in (9). What is the denotation of VP in (10)? And what is the semantic contribution of the rule merging the V and the DP\_ob into the VP?
(9) \( [\text{kiss}]_{M1} = \{<\text{Ann Ann}>, <\text{Ann Betty}>, <\text{Betty Connor}>\} \)
(10) \[
  S \\
  / \ \ / \ \\
  DP_{\text{su}} \ V \ P \ V P \\
  / \ / \ \ \ \\
  \text{Betty} \ V \ D P_{\text{ob}} \\
  / \ \ \ \\
  \text{kisses} \ \text{Connor} \\
  \]
(11) For any arbitrary \( V_{\text{trans}} \), DP and model M
  \[
  [V P]_M = \text{need a set, to make sure rule (8) works!} \\
  / \ \ \\
  V_{\text{trans}} \ D P \\
  \]
• With the 3-place predicate \text{assign}, we have three levels of DP embedding: least embedded (=DP subject), middle embedded (= DP indirect object), and most embedded (= DP direct object). For the least embedded, we can use the rule in (8). For the middle embedded, we can use the rule we gave in (11). But we need a new rule combining the most embedded DP (=the DP direct object) with the ditransitive verb \text{assign}.  

QUESTION 5: What is this rule? And what is the resulting denotation of $V_{shell}$?

(12)\[\begin{array}{c}
S \\
/ \ \ \\
DP_{sw} \ VP \\
/ / \ \\
Ann \ V_{shell} \ to \ DP_{IO} \\
/ \ \ \ \\
V \ DP_{DO} \ Connor \\
/ \\
assigned \ Ann
\end{array}\]

(13) For any arbitrary $V_{dtrans}$, DP and model M
\[
\llbracket V_{shell} \rrbracket_M = \\
/ \ \\
V_{dtrans} \ DP
\]

Problem 4: No way to make complicated kinds of predicates in Predicate Logic

boy can translate to B (a one-place predicate), love can translate to L (a two-place predicate), but what about boy who loves Mary? or who loves Mary, just the relative clause by itself?

4.2 The solution: type theory and lambda calculus
Step 1: Type theory

Individuals – type $e$ \hspace{1cm} John, the red apple, water, the saber-tooth tiger

Truth values – type $t$ \hspace{1cm} The morning star is the evening star, Every boy runs

Sets vs. characteristic functions of sets

• What is the type of sets? \hspace{1cm} red, runs
• The meaning of John runs is a truth value, while John denotes an individual
• We can think of runs as a function that takes an individual to give a truth-value:

Sets can be thought of in terms of their characteristic functions $f_A(a) = 1$ iff $a \in A$

The type of any function that takes arguments of type $i$ to yield a result of type $j$ is $i \rightarrow j$

So, the type of runs is $e \rightarrow t$.
What about the type of kiss or like?
This creates a system of types: $D_e$ – subsets of it are all sets containing individuals
“D” stands for “domain” $D_t = \{0,1\}$
$D_{e+}$ – set containing all the functions from $D_e$ to $D_t$
$D_{et-et}$ – set containing all the functions from sets to sets

QUESTION 6: what English words could have this last type?

Benefits:
- side-steps Russel’s paradox
- neat assignment of semantic categories

Step two: Lambdas

The lambda notation
- Instead of saying, run is a function looking for individuals of type $e$ to make a sentence (of type $t$) saying that this individual runs,
- we can write it down:
(14) $\text{run} \rightarrow \lambda x_e. \text{run}(x)$

Translating lambdas into English: $\lambda X_j. \text{blah}$ translates into
“Give me something of type $j$, and I’ll give you blah”

This is just building Type theory into Predicate Logic.

QUESTION 7:
(15) Give lambda-terms corresponding to the verbs kiss, give, adjective red.

(16) What about quantifiers like some and every?

With this, we can extend Predicate Logic to Typed Lambda Calculus.

Reducing semantic rules to functional application (and maybe some others)
- The basic vocabulary is very similar Predicate Logic, but has an extra symbol $\lambda$:
  1. The lambda operator $\lambda$
  2. Any number of variables for each type
  3. A (possibly empty) set of constants for each type
  4. The logical connectives $\neg$, $\land$, $\lor$, $\rightarrow$
  5. The quantifiers $\forall$ and $\exists$
  6. The equality symbol $=$
  7. parenthesis
The formal lambda-calculus system is based on two operations: *lambda-abstraction*, and *function application*.

- **Lambda abstraction**

  (17) Given any lambda-term \( M \) of type \( j \), \( \lambda X. M \) is a lambda term of type \( i \to j \).

  In our system, we will use mostly this subcase: if \( \phi \) is a well-formed formula, and \( x \) a variable of type \( i \), \( \lambda x. \phi \) is a lambda-term of type \( i \to t \).

- **Function application**

  (18) Given a lambda term \( M \) of type \( j \) and a term \( N \) of type \( i \), \( \lambda X. M \ (N) \) is a term of type \( j \)  
  
  In other words, if \( \phi \) is a term of type \( i \to j \) and \( N \) of type \( i \), then \( \phi(N) \) is of type \( j \)  
  
  Just writing the argument \( N \) next to the function \( M \) doesn't make the formula simpler. So, function application comes with this helpful simplification rule, **\( \beta \)-conversion**:  
  - **Definition of \( \beta \)-conversion**  
    (19) \( \lambda X. M \ (N) = M' \) which is \( M \), but with every occurrence of \( X \) replaced by \( N \).  
  
  Example: \( \lambda x. run(x) \ (j) = \beta \)-conversion = \( run(j) \)

- **Important points:**  
  - Applying \( \beta \)-conversion blindly may lead to trouble:  
    (20) \( \lambda x. \lambda y. love(x, y) \ (y) \to \beta \)-conversion \( \to \lambda y. love(y,y) \) - wrong result!  
  
    (21) Even worse: \( \exists y (x \neq y) \) \( \forall w,g \) = property of being non-unique. (function which will give 1 (True) for every individual as long as \( D_e \) has more than one individual in it)  
  
    \( \exists y (x \neq y) \) \( \forall w,g \) = the value assigned by this function to \( g(y) \), so 1 as long as \( D_e \) has more than one individual in it  
  
    \( \exists y (y \neq y) \) \( \forall w,g \) = 0, since it asserts that there is a thing not equal to itself !!!

  - To avoid that, when we’re substituting \( N \) in \( \lambda X. M \ (N) \), we have to check that **all free variables** in the expression we’re plugging in (here, \( N \)) are **free for \( X \) in \( M \)** – that is,  
    **for every free \( Y \) in \( N \), no free occurrence of \( X \) in \( M \) is in the scope of \( \exists Y \) or \( \forall Y \) or \( \lambda Y \).**

  When this condition is violated (as in examples above), replace a formula with its alphabetic variant (**\( \alpha \)-conversion**).  
  - **Definition of \( \alpha \)-conversion**:  
    (22) If \( Y \) is free for \( X \) in \( M \), and is NOT free in \( M \), then \( \lambda X. M = \lambda Y. [Y/X] \) \( M \)  
    \( \lambda x. \lambda y. love(x,y) \ (y) = \alpha \)-conversion= \( \lambda x. \lambda z. love(x,z) \ (y) = \beta \)-conversion= \( \lambda z. love \ (y,z) \)
• Since all the lambda-terms are either complete formulas or functions, all the semantic rules we saw so far could be stated in just one rule:

**Function application (for semantics of natural language)**

(23) If \( \alpha \) is an expression of type \( i \), \( \beta \) is an expression of type \( i \rightarrow j \), then \( \llbracket \gamma \rrbracket = \llbracket \beta \rrbracket (\llbracket \alpha \rrbracket) \),

\[
\begin{array}{c}
\text{β} \\
\text{α}
\end{array}
\]

which is an expression of type \( j \) resulting from applying the function \( \beta \) to the argument \( \alpha \).

Examples:

(24) a. John runs
    b. Ann assigns Betty to Connor
    c. Everyone runs