2. Background: Sets and Functions

Before we go ahead and study the meaning of linguistic expressions following the outlines of the previous section we should become familiar with some important theoretical tools, in particular with the notions of a set, a relation, and a function, and the various ways to describe them, like the lambda notation.

2.1. Elementary Set Theory

2.1.1. The Notion of Sets

Set theory was developed by the mathematician Georg Cantor at the end of the 19th century as a foundation of mathematics. The underlying idea is to have a basic, simple and consistent theory on which all of mathematics can be built. This is the classical definition of a set:

(1) A **set** is an abstract collection of distinct objects of our experience or thought.

Let us analyze the elements of this definition:

- "abstract": The objects don't have to be physically brought together, that is, not like stamps in a stamp collection.
- "collection": It must be clear which objects belong to it, and which don't. Also, it is just a collection, not an arrangement -- the order in which we list the elements is irrelevant. (Structures in which the order of enumeration is relevant are called **tuples** or **lists**.)
- "distinct": It must be possible to identify the objects, that is, to keep them apart. In particular, one and the same object cannot occur twice in a set. Structures that allow for multiple occurrence of objects are called **multisets**.
- "experience"/"thought": The objects can be concrete (like the students in LIN 380M) or abstract (like the seven cardinal virtues, or the numbers between 3 and 17).

Some examples and further notions:

- The objects that belong to a set are called its **elements**. Nothing is required of the elements — they can be concrete or abstract. In particular, they can be sets themselves.
- Sets may be small (like the set of natural numbers between 3 and 17) or big (like the set of numbers between 3 and 17 billion). These sets are **finite**, but there may be **infinite** sets (e.g., the set of all the natural numbers 1, 2, 3, 4, ...).
- Sets may have only one member (so-called **singletons**); note that the SET that just contains Manfred Krifka has to be distinguished from Manfred Krifka himself (e.g., the set is abstract, I am hopefully concrete).
- Sets may have no member at all (the so-called **empty** set,).

Some notational conventions:

- Sets are typically given by capital letters A, B, C, X, Y etc.
- Elements are typically given by small letters a, b, c, x, y etc.
- a A "a is an element of A", a A"a is not an element of A"

When are two sets equal?

- We would at least require that if set A and set B are equal, then A and B have the same elements.
- We can even claim that if A and B have the same elements, then they are equal, that is, only membership counts for the identification of sets. (Sets differ in that respect from committees, music bands etc. It may be that John, Bill and Mary play rock music as 'The Rocking Stones'

on Friday nights, and play country as 'The Rattlesnakes' on Saturday nights; of course, these two bands have quite different properties, even if they consist of the same members!).

- Hence: Two sets are equal iff they contain the same elements.

2.1.2. How to Specify Sets: Enumeration and Abstraction

There are several ways in which sets can be given. Here we will talk about two methods: Enumeration and Abstraction.

Enumeration

We may simply **enumerate** the elements of a set. This is also called the "list notation".[1] We do this conventionally by using braces and separating the elements by commas. Examples:

(2) a.{Haydn, Mozart, Beethoven} (let's call this set of composers ℂ).
b.{Mozart, Sirius, 3}
c.{Haydn, {Mozart, Beethoven}} (this is different from ℂ)
c.{Mozart} (a singleton set)
d.{} (the empty set, also written Ø)

Note:

- The order in the enumeration is irrelevant: {Mozart, Beethoven, Haydn} = ℂ
- The number of occurrences of an element is irrelevant:
{Haydn, Mozart, Mozart, Beethoven} = ℂ

Abstraction

Also called **predicate notation**, abstraction describes the elements that belong to this set in some language -- English or a mathematical language. All the objects that fit to the description, and nothing but these objects, are supposed to be in this set. The following format is used:

{Variable | Description containing the variable}

We typically use x, y, z as letters for variables. Some examples:

(3) a.{x | x is a major composer of the classic Vienna period}
(= our set ℂ)
b.{x | x is an Egyptian pharao}
(= {Cheops, Chefren, … Cleopatra})
c.{x | x is a number between 1 and a billion}
(= {1, 2, … 1,000,000,000})
d.{x | x is an even number}
(= {2, 4, 6, 8, …}, let's call this set 𝔼).

We read (3.a) as "the set of x such that x is a major composer of the classic Vienna period".

What if the variable x does not occur in the description, as in the following examples:

(4) a.{x | Austin is the capital of Texas}
b.{x | Houston is the captal of Texas}

According our definition, (a) is the set of x such that "Austin is the capital of Texas" is true. This description is true, regardless which value x has. This means that everything is in this set. By similar reasoning, nothing is in the set (b).

[1] This term is misleading; for lists the order of the things listed matters, but not for sets.

2.1.3. Cardinality of Sets

Sometimes we are just interested in the "size" of a set, that is, the number of elements in it, its so-called **cardinality**. The cardinality of a set A is given by |A|, or #(A), or #A, or card(A). Examples:

(5) a.#{Mozart, Haydn, Beethoven} = 3
b.#{Mozart, {Haydn, Beethoven}} = 2 (!)
c.#{Mozart, Haydn, Haydn, Beethoven} = 3 (!)
d.#Ø = 0

The notion of cardinality has been extended to infinite sets like $\mathbb{E}$. Of course, the set $\mathbb{E}$ cannot be any specific natural number. For the cardinality of infinite sets like $\mathbb{E}$, mathematicians use the symbol $\aleph_0$ (aleph-zero).[2]

2.1.4. The Subset Relation and Power Sets

Sometimes we want to say that all the elements of one set A are also in another set B. We write A B and say that A is a **subset** of B and B is a **superset** of A. To say that A is not a subset of B, we write A _ B. Examples:

(6) a. $\mathbb{C}$ {Haydn, Mozart, Beethoven, Schubert}
b. $\mathbb{E}$ {x|x is a natural number}
c. $\mathbb{C}$ $\mathbb{C}$

In the last case, the subset relationship holds in both directions. If we want to exclude that and express the **proper** subset relationship, we use the symbol and write A B for "A is a proper subset of B". We can define this notion as follows:

(7) A B iff A B and B _ A.

To express that A is not a proper subset of B we write A B. Examples:

(8) a. $\mathbb{C}$ {Haydn, Mozart, Beethoven, Schubert},
b. $\mathbb{C}$ $\mathbb{C}$

Note that, according to our definition, the empty set Ø is subset of every set A (recall the definition of A as "every element of is an element of A", and observe that this is trivially true, as does not have any element).

(9) Ø A, for every set A.

With singleton sets, we have to distinguish strictly between the subset relationship and elementhood. For example:

(10) a.4 $\mathbb{E}$, but not: 4 E.
b.{4} $\mathbb{E}$, but not: {4} $\mathbb{E}$.

The set of all subsets of a set A is called the **powerset** of A. We write pow(A) for it.

(11) Definition: $\text{pow}(A) =_{\text{def}} \{X \mid X \quad A\}$

How many elements are in a power set? First, look at a few examples:

[2] There is more than one type of infinity — actually, there are infinitely many different types of infinity. Unfortunately, our time in this course is too limited to go into that.

(12) a.pow({a, b, c}) = {{a, b, c}, {a, b}, {b, c}, {a, c}, {a}, {b}, {c}, } (set: 3 elements, powerset: 8 elements)
b.pow({a, b}) = {{a, b}, {a}, {b}, }
(set: 2 elements, powerset: 4 elements)
c.pow({a}) = {{a}, }
(set: 1 element, powerset: 2 elements)
d.pow( ) = { }
(set: 0 elements, powerset: 1 element)

In general, if a set A has n elements, we have 2 2 2 (n times) elements in the powerset of A.

(13) cardinality of power sets: If #(A) = n, then #(pow(A)) = 2^n.

2.1.5. Set-theoretic Operations

There are a number of important operations that can be performed with sets. The **union** of two sets A, B, written A B, is the set that contains all and only the elements contained in A or B:

(14) A B = $_{def}$ {x| either x A or x B}

For example,

(15) a.{1,2,3} {3,4,5} = {1,2,3,4,5}
b.{1,2,3} {2,3} = {1,2,3}
c.{1,2,3} Ø = {1,2,3}

The **intersection** of two sets A, B, written A ∩ B, is the set that contains all and only the elements contained in both A and B:

(16) A ∩ B = $_{def}$ {x| both x A and x B}

For example,

(17) a.{1,2,3} ∩ {3,4,5} = {3}
b.{1,2,3} ∩ {7,8,9} = Ø
c.{1,2,3} ∩ Ø = Ø

The set-theoretic **difference** A ¾ B is the set that contains all and only the elements of A that are not contained in B:

(18) A ¾ B = $_{def}$ {x| x A and x B}

For example,

(19) a.{1,2,3} ¾ {3,4,5} = {1,2}
b.{1,2,3} ¾ {7,8,9} = {1,2,3}
c.{1,2,3} ¾ Ø = {1,2,3}

Notice that the set-theoretic operations , ∩ and — are quite different from the subset relation. When we write, for example, A B, we have a new set. Wen we write A B we don't have a new set, but a statement that can be true or false.

Often, we restrict our attention to a certain set of elements, e.g. the set of all natural numbers, and look at subsets of this set. Such a set is called **universe**, often denoted by U. With respect to a universe U, we define the **complement** of a set A (where A U), written A′, as follows:

(20) A′ = $_{def}$ U ¾ A.

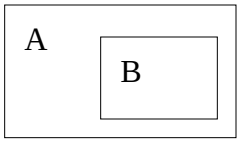
For example, with respect to the natural numbers as a universe,

(21) a. $\{1,2,3\}' = \{x \mid x \text{ is a natural number and } x \geq 4\}$
b. $\mathbb{E}' = \{x \mid x \text{ is an odd number}\}$

2.1.6. Venn Diagrams

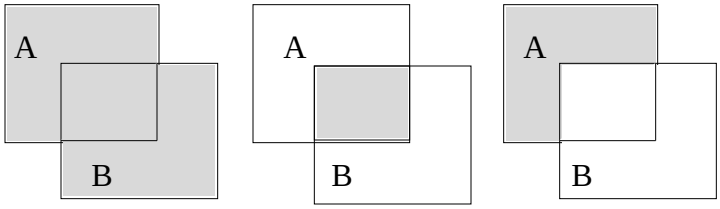
A handy way to depict set-theoretic relationships is by so-called **Venn diagrams** (after the mathematician John Venn), also called "Euler circles" (after the 18th century mathematician Leonhard Euler, who invented them as a didactic device in his *Lettres à unde Princesse d'Allemagne*, published in St. Petersburg in 1768). We already have used something like Venn diagrams in the previous chapter. In a Venn diagram, the elements are represented by the points in a plane, and sets of elements are represented by enclosed areas. Examples:

(22)



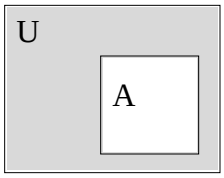
B A

(23)



A B A∩B A — B

(24)



A′ (Complement of A, = U — A)

2.1.7. Set-Theoretic Laws

With set-theoretic notions like union, intersection, subset, complement etc., we find that they follow nice structural patterns that we can express by certain equations. In the following, X, Y, Z are supposed to be arbitrary sets. The laws can be illustrated by Venn diagrams.

- **Idempotency**: X X = X X∩X = X
- **Commutativity:** X Y = Y X X∩Y = Y∩X
- **Associativity:** (X Y) Z = X (Y Z) (X∩Y)∩Z = X∩(Y∩Z)

By associativity, we see that the bracketing in an expression that contains either just or just ∩ does not matter. We may write X Y Z for (X Y) Z.

Also, we can generalize the notation for union and intersection in the following way. Assume that X is a set of sets, then:

(25) a. X =$_{\text{def}}$ the union of all sets in X.
b. ∩X =$_{\text{def}}$ the intersection of all sets in X.

Examples:

(26) a. $\{\{a,b,c\}, \{c,d,e\}, \{e,f,g\}\} = \{a,b,c,d,e,f,g\}$
b. $\cap\{\{a,b,c\}, \{b.c.d\}, \{c.d.e\}\} = \{c\}$

- **Distributivity:** X (Y∩Z) = (X Y)∩(X Z)
  X∩(Y Z) = (X∩Y) (X∩Z)
  that is, "distributes" over ∩, and vice versa.
- **Identity laws:** X Ø = X X∩Ø = Ø
  X U = U X∩U = X
  that is, the empty set Ø and the universe U
  play a special role for union and intersection.
- **Complement laws:** X X′ = U X∩X′ = Ø
  (X′)′ = X X¾Y = X∩Y′
- **de Morgan laws:** (X Y)′ = X′∩Y′
  (X∩Y)′ = X′ Y′
- **Consistency:** X Y iff X Y = Y X Y iff X∩Y = X

Consistency allows us to define by or ∩, and to define or ∩ by .

The set-theoretic laws allow us to simplify certain expressions. Example:

(27) (A B) (B∩C)′ =
(A B) (B′ C′) = (de Morgan)
A (B (B′ C′)) = (Associativity)
A ((B B′) C′) = (Asscociativity)
A (U C′) = (Complement)
A U = (Identitity)
U (Identitity)

Such a reduction of one expression to another, usually simpler one is called a **proof**: It's a sequence of expressions where each expression can be derived from its predecessor by application of a law. Consider another example, in which we show that a certain fact holds:

(28) a. Show that A∩B A B!
b. Consistency says: X Y iff X∩Y=X;
with X: (A∩B) and Y: (A B), this means (A∩B)∩(A B) = A∩B.
c. To prove this we can use set-theoretic laws:
(A∩B)∩(A B)
((A∩B)∩A) ((A∩B)∩B) (Distributivity)
(A∩(A∩B)) ((A∩B)∩B) (Commutativity)
((A∩A)∩B) (A∩(B∩B)) (Associativity, twice)
(A∩B) (A∩B) (Idempotency, twice)
A∩B (Idempotency)
q.e.d. ("quod erat demonstrandum").

2.1.8. Set Theory and Semantic Relations

Set theory has a wide area of application. For example, we can use it to model **word meanings**. Let the meaning of a noun, a predicative adjective, or an intransitive verb α be the set of objects to which α applies. Recall that we give the meaning of α by ⟦α⟧. For example,

(29) a. ⟦*horse*⟧ = {x| x is a horse}
b. ⟦*red*⟧ = {x| x is red}
c. ⟦*talk*⟧ = {x| x talks}

The relation of **hyponymy** can be expressed by the subset relation: ⟦*horse*⟧ ⟦*animal*⟧

Combinations of expressions by *and, or* and *not* (the so-called **Boolean operators**) can be expressed by set union, intersection, and complement:

(30) a. ⟦*red and round*⟧ = ⟦*red*⟧ ∩ ⟦*round*⟧
b. ⟦*red or round*⟧ = ⟦*red*⟧ ⟦*round*⟧
c. ⟦*not red*⟧ = ⟦*red*⟧′

The set-theoretic laws predict the semantic behavior:

(31) a. ⟦*red and round*⟧ = ⟦*round and red*⟧, as
⟦*red*⟧ ∩ ⟦*round*⟧ = ⟦*round*⟧ ∩ ⟦*red*⟧
(Commutativity)
b. ⟦*red and* [*round and soft*]⟧ = ⟦[*red and round*] *and soft*⟧, as
⟦*red*⟧ ∩ (⟦*round*⟧ ∩ ⟦*soft*⟧) = (⟦*red*⟧ ∩ ⟦*round*⟧) ∩ ⟦*soft*⟧
(Associativity)
c. ⟦*not* [*red and round*]⟧ = ⟦[*not red*] *or* [*not round*]⟧, as
(⟦*red*⟧ ∩ ⟦*round*⟧)′ = ⟦*red*⟧’ ⟦*round*⟧′
(de Morgan)
d. ⟦*red and* [*round or soft*]⟧ = ⟦[*red and round*] *or* [*red and soft*]⟧, as
⟦*red*⟧ ∩ (⟦*round*⟧ ⟦*soft*⟧) = (⟦*red*⟧ ∩ ⟦*round*⟧) (⟦*red*⟧∩⟦*soft*⟧)
(Distributivity)

So, set theory can be used to express theories about the semantics of natural languages — both about the semantic relationships between words and the semantic relationships between complex expressions.

2.2. Functions

Another fundamental concept in mathematics are **functions**. Functions can be seen as a special type of **relations**, to which we turn first.

2.2.1. Ordered Pairs and Relations

With sets we can model simple predicates like *red, ball*, or *run.* But what about transitive verbs, like *love*, or nouns like *father (of)*? Notice that they don’t hold of single objects, but of a **pair** of two objects. That is, they are **relations** between two entities. Examples:

(32) a.Mary loves John.
b.Abraham is the father of Isaac.

Do we need a completly new concept for relations, or can we do it within set theory? We may, for example, try to model relations are sets of sets of two elements:

(33) ⟦*father*⟧
= {{Isaac, Abraham}, {Jacob, Isaac}, {Esau, Isaac}, ...}
= {{x,y}| y is the father of x}

There is a problem here: Many relations are order-sensitive, but sets aren’t. According to the above definition, Isaac would also count as a father of Abraham.

A second attempt: Relations as sets of **ordered pairs**. Ordered pairs are similar to sets with two elements, except that the order in which they are listed matters. We write ordered pairs thus:[3]

Isaac, Abraham

We must make sure that a pair like Isaac, Abraham differs from a pair like Abraham, Isaac . This follows if we assume the following identity criterion for ordered pairs:

(34) Two ordered pairs x,y , u,v are equal iff x=u and y=v.

With the notion of ordered pairs we can give the following interpretation:

(35) ⟦*father*⟧ = { Isaac, Abraham , Jacob, Isaac , Esau, Isaac , …}
= { x,y | y is the father of x}

There are various ways to indicate that two elements x, y stand in a relation R. Any one of the following notations will do:

(36) a. x, y R
b.xRy
c.R(x,y)

The notion of all pairs that can be formed out of two sets, taking the first elements out of one set and the second elements out of the other, is called the **Cartesian Product**:

(37) If A, B are two sets, then A×B = { x, y | x A, y B}.

For example,

(38) {a, b, c, d} × {1, 2, 3} = { a, 1 , a, 2 , a, 3 ,
b, 1 , b, 2 , b, 3 ,
c, 1 , c, 2 , c, 3 ,
d, 1 , d, 2 , d, 3 }

A relation R between a set A and a set B is a subset of the Cartesian product A×B.

Once we have defined the notion of an ordered pair, we can easily define **triples**, **quadruples**, etc., in general: **n-tuples:**

(39) a.triples: x, y, z $=_{def}$ x, y , z (reduction to pairs)
b.quadruples: u, x, y, z $=_{def}$ u, x, y , z (reduction to triples)
c.n-tuples: $x_1, x_2, \ldots x_n =_{def} x_1, x_2, \ldots x_{n-1}, x_n$
(reduction to (n-1) tuples)

Again, the expression on the left-hand side can be seen as an abbreviation of the expression on the right-hand side. We will need triples for the semantic description of ditransitive verbs like *give*, which we may render as follows:

(40) ⟦*give*⟧ = { x, y, z | x gives y to z}

---

[3] One question that arises at this point is: Do we have to introduce ordered pairs as a new kind of mathematical entity, or can we reconstruct or encode it with the help of sets? It turns out that he latter is true. We can define ordered pairs with the notion of sets. This is the standard reconstruction, by Wiener and Kuratowski:

x,y $=_{def}$ {{x}, {x,y}}

From the representation {{x}, {x,y}}, we can derive the first member of the pair as the element in the singleton set {x}, namely, x, and the second member as the element of the set that we obtain when we subtract the singleton set from the other set, {x,y} ¾ {x} = {y}, that is, y. Notice that this reconstruction of ordered pairs by sets satisfies the criterion of identity for ordered pairs:

{{x},{x,y}} = {{u},{u,v}} iff x=u and y=v.

In particular, we have {{x},{x,y}} ≠ {{y},{x,y}} (if x ≠ y). We can see the notation x,y as an abbreviation of the underlying notation {{x}, {x,y}}.

We call a set of pairs a **two-place** relation, and a set of triples a **three-place** relation. (A simple set of entities then could be called a **one-place** relation).

2.2.2. Functions as Relations

Relations are pairings between objects; for example, the relation ⟦*father*⟧ pairs every person x with x's father. This relation has an important property: Every person has exactly one father. We say that such relations have the property of **right-uniqueness**, defined as follows:

(41) A relation R is right-unique iff the following holds:
For every x, y, z, if x, y R and x, z R, then y = z.

Clearly, ⟦*father*⟧ is right-unique. The following relations are not right-unique:

(42) a. { x, y | y is the son of x}
(notice that Jacob and Esau have the same father!)
b. { x, y | x, y are numbers, and y is greater than x}

Relations that are right-unique are called **functions**. The property of right-uniqueness allows for a new type of notation that you will recall from your math class. Instead of writing x,y R or one of its equivalents, we can write the following:

(43) y = R(x).

We say that x is the **argument**, and y is the **value**. We also say that R is **applied** to x, and we say that R **maps** x to y. We often use letters like f, F, g, G for functions. Notice that this notiation is impossible for relations that are not functions. We can say that the father of Jacob IS Isaac, but we cannot say that the son of Isaac IS Jacob — after all, Isaac had a second son, Esau.

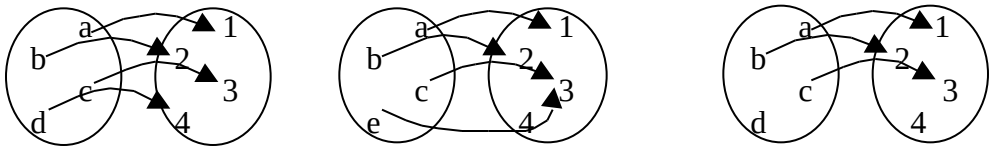
Another important pair of notions is the **domain** and the **range** of a function f:

(44) a. The domain of a function f, DOM(f)
= {x | there is a y such that x, y f}
b. The range of a function f, RNG(f)
= {y | there is a x such that x, y f}

That is, the domain of f are the possible arguments of f, and the range of f are the possible values of f. The domain of ⟦*father*⟧ is the set of persons (everyone has a father), and the range of ⟦*father*⟧ is the set of fathers.

When A is the domain of f and B its range, we say that f is a function from A **onto** B. If B is a subset of C, then we say that f is a function from A **into** C. (Notice that, technically, A is also a function **into** B). If f is a function from A into C, we write f: A → C for that. And if A is a subset of C, then we say that f is a **partial function** from C into B. (Notice that, technically, f is also a partial function from A onto B).

(45)



function onto/into function into partial function into

It is sometimes convenient to list functions with the following notation:

(46) ⟦*father*⟧ = { Isaac → Abraham, Jacob → Isaac, Esau → Isaac, … }

2.3. The Lambda Notation

The lambda calculus is a very perspicuous way to work with functions that we will use throughout this course. This is a practical introduction to the notation, and some of its applications. I will mostly use mathematical examples here, but we will also see how it can be applied to express meanings.

2.3.1. How To Describe Functions

Functions can be described by enumerating their argument-value pairs, either as a set of pairs or in the notion illustrated with (46). Often such an explicit listing is not very handy or even impossible. In this case we can **describe** a function. One format is the following:

(47) a. ⟦*mother*⟧ m: persons → persons,
x ↦ the mother of x.
b. ⟦*successor*⟧ s: natural numbers → natural numbers,
x ↦ x + 1
c. ⟦*square*⟧: q: natural numbers → natural numbers,
x ↦ x^2

The function f is defined as a function from persons into persons, where f maps every x to the father of x. We have, for example, f(Isaac) = Abraham. And the function s is defined as a function from natural numbers to natural numbers, where s maps every natural number x to the successor of x, that is, the number x + 1. We have, for example, s(13) = 14.

You will no doubt be familiar with another notation for functions, which is often used in algebra:

(48) $f(x) = x^2 + x + 1$

This is a function f that maps every number x to $x^2 + x + 1$. For example, we have f(2) = 7.

In the formats illustrated so far we have defined the functions like m, s, q, f. That is, whenever we use f and s, for example in expressions like m(Isaac) or f(13), we have to look up how these functions were defined. The lambda notation gives us a way to name functions in a way that makes it immediately clear how they are defined. In this notation, f and s would be rendered as follows:

(49) m: λx[the mother of x]
s: λx[x+1]
q: λx[x^2]
f: λx[$x^2 + x + 1$]

We call such expressions **lambda terms**. The structure of lambda terms is as follows:

(50) λ variable [description of value of variable]

This stands for the function that maps the things that 'variable' can possibly stand for to the value of the variable, as described in the so-called body of the lambda term. The formation of a lambda term from a description containing a variable, like "mother of x", or "$x^2 + x + 1$", is called (lambda) **abstraction**.

As already said, a neat property of the lambda notation of functions is that they wear their definition on their sleeves. This makes it possible to give the value of the function applied to an argument by simply replacing the lambda variable by that argument. This is called (lambda) **conversion** or **reduction**:

(51) a. λx[mother of x](Isaac)
= mother of Isaac
= Sarah

b. λx[x^2 + x + 1](3)
= 3^2 + 3 + 1
= 9 + 3 + 1
= 13

The lambda notation, as developed so far, is less flexible than some other notations. This is because it does not make explicit the domain of the function. In our current version of the lambda notation, the two functions

(52) a. f: $\mathbb{N}$ → $\mathbb{N}$
x ↦ x^2 + x + 1

b. g: $\mathbb{E}$ → $\mathbb{N}$
x ↦ x^2 + x + 1

(52.a) is a function from natural numbers to natural numbers, whereas the domain of (b) is more restricted, it is a function from even numbers. Our current notation does not distinguish between those two functions. But we can easily extend it, in one of the following ways:

(53) a. λx $\mathbb{E}$ [x^2 + x + 1]
b. λx[x $\mathbb{E}$ | x^2 + x + 1]

We will use this notation whenever it is relevant to specify the domain of a function, which will often be the case. Of course, if a function is applied to an entity that is not in its domain, its value is not defined. There is an important difference between the following functions:

(54) a. λx[x $\mathbb{N}$ | x^2 + x + 1](3) = 13
b. λx[x $\mathbb{E}$ | x^2 + x + 1](3): not defined.

2.3.2. Functions as Arguments, and Functions that don't have exactly one Argument.

In the examples considered so far the argument that we applied a function to always was something simple — a number, or a person. But functions can expect complex arguments, for example, sets or other functions. This is often indicated with different variable names, like X for sets or f for functions.

(55.a) is a function that takes a set and intersects it with the set {1, 2, 3}. Applied to the set {2, 3, 4}, we get the set {2, 3}; applied to the set {4, 5, 6}, we get the empty set, and the function is not defined for a person, like Isaac.

(55) a. λX[X ∩ {1, 2, 3}]
b. λX[X ∩ {1, 2, 3}]({2, 3, 4}) = {2, 3}
c. λX[X ∩ {1, 2, 3}]({4, 5, 6}) =
d. λX[X ∩ {1, 2, 3}](Isaac): undefined

The function is not defined when applied to a person because intersection ∩ is only defined for sets. We could, of course, make this explicit in our extended lambda notation:

(56) λX[X is a set | X ∩ {1, 2, 3}]

In the following example we have a function that takes a function and gives a value as a result — the result that we get when we apply this function to the number 3.

(57) λf[f(3)]

We can work with such higher-order functions as usual. As before, we replace the variable, f, by the argument. Typically we can reduce the resulting description of the value even further, by applying the function f to an argument, here 3:

(58) a. $\lambda f[f(3)](\lambda x[x^2])$
$= \lambda x[x^2](3)$
$= 9$

b. $\lambda f[f(3)](\lambda x[x^2 + x + 1])$
$= \lambda x[x^2 + x + 1](3)$
$= 13$

A more complex example of a higher-order function, with an example:

(59) a. $\lambda f[f(3) + f(4)]$
b. $\lambda f[f(3) + f(4)](\lambda x[x^2 + x + 1])$
$= \lambda x[x^2 + x + 1](3) + \lambda x[x^2 + x + 1](4)$
$= 13 + 21$
$= 34$

And one more example, just to get the hang of it. In the derivation I underline the part that is dealt with in the following line.

(60) a. $\lambda f[f(f(3)–9)]$
b. $\lambda f[f(f(3)–9)](\lambda x[x^2 + x + 1])$
$= \lambda x[x^2 + x + 1](\underline{\lambda x[x^2 + x + 1](3)} – 9)$
$= \lambda x[x^2 + x + 1](\underline{13 – 9})$
$= \underline{\lambda x[x^2 + x + 1](4)}$
$= 21$

To compute such functions is not really difficult; it is just easy to get a bit confused. It is something in which computers are better than humans, of course. Indeed, one of the first programming languages, LISP, was based on the lambda calculus, and it is still widely used, especially in Artificial Intelligence research.

With functions that take functions as arguments we must watch out for cases in which the function is not defined, which may turn out to be the case quite late in the derivation. Consider the following example:

(61) $\lambda f[f(f(4)–8)](\lambda x[\ x \in \mathbb{E} \mid x^2 + x + 1])$
$= \lambda x[\ x \in \mathbb{E} \mid x^2 + x + 1](\underline{\lambda x[\ x \in \mathbb{E} \mid x^2 + x + 1](4)} – 8)$
$= \lambda x[\ x \in \mathbb{E} \mid x^2 + x + 1](\underline{21 – 8})$
$= \lambda x[\ x \in \mathbb{E} \mid x^2 + x + 1](13)$: undefined

In the last step it turns out that the function, 13, is not defined for the argument because it is not an even number. Hence the original function $\lambda f[f(f(4)–8)]$ applied to the argument $\lambda x[\ x \in \mathbb{E} \mid x^2 + x + 1]$ is undefined.

We have seen that we can define functions that take functions as arguments; now we will consider cases of functions that deliver functions as values. One example is the following:

(62) $\lambda x \lambda y[x^2 + y]$

This function takes a value x and delivers a function that in turn takes a value y and gives the value $x^2 + y$. Consider the following example, where we first apply the function $\lambda x \lambda y[x^2 + y]$ to the argument 3, which gives us the function $\lambda y[9 + y]$. Then we apply this function to the argument 4.

(63) $\underline{\lambda x \lambda y[x^2 + y](3)}(4)$
$= \lambda y[9 + y](4)$
$= 9 + 4$
$= 13.$

This is perhaps a good place to talk about the naming of variables. In principle, the name of a variable in the description of a function should not influence the meaning of this description. After all, variables are just placeholders, and it should not matter how the placeholders are called. The following two expressions describe exactly the same function:

(64) a. $\lambda x[x^2 + x + 1]$
b. $\lambda y[y^2 + y + 1]$

With functions that yield functions as values, we typically have to distinguish the variables. While we can exchange variables and still get the same function, we cannot collapse distinctions made by different variable names.

(65) a. $\lambda x \lambda y[x^2 + y]$
b.= $\lambda y \lambda x[y^2 + x]$
c. ≠ $\lambda x \lambda x[x^2 + x]$

Does (65.c) have any meaning at all? Indeed, it has. The official rules how the lambda notation should be interpreted is based on the notion of **scope** of a variable. This is the part of the description in which occurrences of the variable have to be replaced by the argument. In general, we have:

(66) In a lambda term λ v […], the part "[…]" is the scope of "v".

Scope is defined with respect to a particular occurrence of a variable (a **token**), and not in general for a variable name (or, a **type**). In the following examples, I identify tokens of variables by primes, if necessary.

(67) a. $\lambda x[x^2 + x + 1]$ b. $\lambda x[3x + \lambda x'[x'^2 + x' + 1](x)]$

scope of λx scope of $\lambda x'$

scope of λx

The convention is that a variable in a description is in the scope of the nearest lambda operator with the same variable. We say that the lambda operator **binds** that variable. While there is nothing wrong with lambda expressions like (65.c) or (67.b), it is much more perspicuous to use different variables, such as the following:

(68) a. $\lambda x \lambda y[y^2 + y]$
b. $\lambda x[3x + \lambda y[y^2 + y + 1](x)]$

Notice that, in the case of (68.a), the variable x does not appear at all in the body of the lambda term. This is not a problem; it shows rather that we have a constant function. For example, this is a function that maps everything to the number 13:

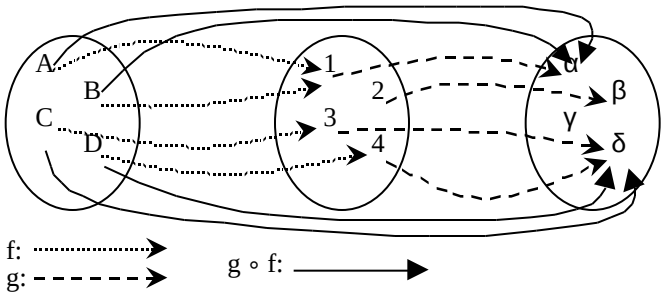
(69) $\lambda x[13]$

As the particular name of a variable does not matter, as soon as it is distinct from other variables that are bound by other lambda operators, we can freely rename variables, provided that we do this in a consistent way. This is often necessary to make a lambda term more readable, or to proceed with the computation of values.

2.3.3. Function Composition

We can not only apply functions to arguments; we can also compose functions (and more generally, relations). For example, the composition of the function f: {A, B, C, D} → {1, 2, 3, 4} and the function g: {1, 2, 3, 4} → {α, β, γ, δ} is indicated in the following diagram.

(70)



We write g∘f for the composition of the function f with the composition g. Why in this order, which first seems counter-intuitive? Because then the following holds, in general:

(71) For all x in the domain of f: g(f(x)) = g∘f(x)

But we don't need a new symbol for function composition. We can define it with the help of lambda terms, as follows:

(72) Function composition: λf[λg[λx[f(g(x))]]]

This lambda term takes a function f and then a function g, and yields the function composition g∘f, which we can describe as λx[f(g(x))].

Take the following example, a composition of the successor function and the doubling function, for natural numbers:

(73) λf[λg[λx[f(g(x)))]]](λy[y + 1])(λz[z + z])
= λg[λx[λy[y + 1](g(x))]](λz[z + z])
= λx[λy[y + 1](λz[z + z](x))]
= λx[λy[y + 1](x + x)]
= λx[x + x + 1]

Question: Do we get the same result for the composition of the doubling function with the successor function?

Function composition may appear of purely mathematical interest. It is not, however. For example, the meaning of *maternal grandfather* can be defined as a composition of the meaning of *father* and the meaning of *mother*:

(74) ⟦*maternal grandfather*⟧
= λf[λg[λx[f(g(x)))]]](λy[the father of y])(λz[the mother of z])
= λg[λx[λy[the father of y](g(x))]](λz[the mother of z])
= λx[λy[the father of y](λz[the mother of z](x))]
= λx[λy[the father of y](the mother of x)]
= λx[the father of the mother of x]

Some languages have a fairly transparent system for these relations. An example is Swedish, which has *morfar*, lit. 'mother father', for *maternal grandfather* (and similarly *mormor, farmor* and *farfar*). The meaning of the complex term *morfar* then obviously is derived by the meanings of the simpler terms *mor* and *far* by functional composition:

(75) ⟦*morfar*⟧ = λf[λg[λx[f(g(x)))]]](⟦*far*⟧)(⟦*mor*⟧), = ⟦*far*⟧ ∘ ⟦*mor*⟧

2.4. Characteristic Functions

We have introduced the notion of sets. Based on this notion we have introduced relations, and functions as a particular type of relations. Now, functions can be used to develop a new notation for sets.

2.4.1. Truth Values Functions

We have defined sets as collections of elements (of a given universe). To know a set means, to be able to identify the elements of that set. That is, we have to know, for each object, whether it is in the set or not. This information can be given as a function, namely, as a function that maps every object in the universe to one of two values:

- to the value True (1), if the element is in the set;
- to the value False (0), if the element is not in the set.

Here, True and False are the two **truth values** we have discussed in the introduction. We often use the numbers 1 and 0 for the truth values.

Such functions are called **characteristic functions** of a set, as they "characterize" the set. We write χ_A ("Chi-A") for the characteristic function of set A. Given a universe U, and a set A, A U, we have the following definition of the characteristic function of A:

(76) χ_A: U $\rightarrow$ {0,1},
x $\mapsto$ 1 if x A,
x $\mapsto$ 0 if x A.

Example:

(77) Let U, the universe, be the set of small letters {a, b, c, ... z}.
Then we have:
$\chi_{\{b,c\}}$: U $\rightarrow$ {0,1}
a $\mapsto$ 0, b $\mapsto$ 1, c $\mapsto$ 1, d $\mapsto$ 0, ... z $\mapsto$ 0

The notion of a characteristic function allows us to clarify an important relation between the abstraction notation for sets and the lambda notation for functions. Take the following set:

{x | x is a woman}

This defines the set of all women. Its characteristic function is the function that maps every x to 1 if x is a woman, and to 0 if it is not a woman:

(78) $\chi_{\{x \mid x \text{ is a woman}\}}$ U $\rightarrow$ {0, 1}
x $\mapsto$ 1 if x {x | x is a woman}, i.e. if x is a woman
x $\mapsto$ 0 if x {x | x is a woman}, i.e. if x isn't a woman.

2.4.2. Characteristic Functions as Lambda Terms

Can we render characteristic functions as lambda terms? Not quite, because lambda terms do not allow for disjunctive descriptions, like "if x is so-and-so, then the value is this-and-this, but if x is such-and-such, the value is that-and-that". But we can introduce the following convention for descriptions in lambda terms that intuitively are sentences:

(79) In a lambda term "λ variable Domain [description of value]", if the description of value is a sentence that can be true or false, it stands for 1 if the sentence is true for the given argument, and for 0 if the sentence is false for the given argument.

This allows us to render the characteristic function of the set {x | x is a woman} as follows:

λx U[x is a woman]

Instead of writing, e.g., Mary {x | x is a woman}, we can use the function notation and write:

λx U[x is a woman](Mary)
= 1, if Mary is a woman, = 0, if Mary is not a woman.

We can use this notation for mathematical examples as well. For example, instead of using the set notation, {x | x $\mathbb{N}$ and x$\geq$7} (the set of natural numbers greater or equal 7) we can use the lambda notation to define the characteristic function of this set:

$$\lambda x \quad U[x \quad \mathbb{N} \text{ and } x \geq 7]$$

The lambda notation is actually more expressive. The function just given maps everything to 1 if it is a natural number greater or equal 7, and to 0 else. With the lambda notation we can define the following function:

$$\lambda x \quad \mathbb{N}[x \geq 7]$$

This function maps every natural number to 1 if it is greater or equal 7, and to 0 else. If we apply it to something that is not a natural number, it does not give us anything, because it is not in the domain of this function.

Hence the notation using characteristic functions is more expressive than the set notation in the following aspect:

- With the set notation {x | ...x...}, there are two choices for a particular object e:
  Either e {x | ...x...} (that means the description ...e... is true),
  or e {x | ...x...} (that means the description ...e... is not true).
- With the function notation λx D[...x...], there are three choices for a particular object e.
  Either e is in the domain D, and the description ...e... is true,
  or e is in the domain D and the description ...e... is false, or e is not even in the domain of the function.

2.4.3. Set-Theoretic Operations for Characteristic Functions

As characteristic functions are closely related to sets, we can extend the set-theoretic relations of subset and proper subset , and the operations of intersection ∩, union and subtraction —, to characteristic functions. But we have to be careful; we have seen that characteristic functions are more expressive than sets insofar as we can distinguish between the function returning the value 0 for a given object, or the function not even being defined for that object. We will use the set-theoretic notions in the following sense:

(80) If c, d are two characteristic functions, then:
a. c d is defined if DOM(c) DOM(d).
if defined, it is true if {x | c(x)} {x | d(x)}, else false.
b. c d is defined if DOM(c) DOM(d).
if defined, it is true if {x | c(x)} {x | d(x)}, else false.
c. c ∩ d = λx[x DOM(c)∩DOM(d) | x {x| c(x)} ∩ {x | d(x)}]
d. c d = λx[x DOM(c)∩DOM(d) | x {x| c(x)} {x | d(x)}]
e. c — d = λx[x DOM(c)∩DOM(d) | x {x| c(x)} — {x | d(x)}]
f. c′ = λx[x DOM(c) | x {x | c(x)}]

For example, the union of two characteristic functions c and d is a function that is defined for all x that are both in the domain of c and in the domain of d. It maps x to true if x is in the union of the objects that c maps to true and that d maps to true, and else it maps x to false.

2.4.4. Rendering Relations as Functions

We have seen that characteristic functions allow us to render sets as functions. Relations are sets, namely, sets of pairs, and consequently we can render them as functions as well. For example, take the following relation, the meaning of *parent*:

(81) a. ⟦*parent*⟧ as relation:
{ x, y | y is a parent of x}.
b. ⟦*parent*⟧ as function:
λ x,y { x,y | x Person, y Person}[y is a parent of x]

For example, λ x,y [y is a parent of x](Isaac, Abraham) = 1. The first notation allows for statements like the following:

Isaac, Abraham { x, y | y is a parent of x},
as Abraham is a parent of Isaac.

For the second notation, we have to allow for variables that come in pairs, and then we can make use of functional application to an argument that is itself a pair. This allows for functional applications like the following:

λ x,y [y is a parent of x](Isaac, Abraham) = 1,
as Abraham is a parent of Isaac.

But there is also another way to make use of the functional notation. Instead of treating *parent* as denoting a function over pairs, we can take it as a function that first takes one argument (say, the child), and then another. (This reduction to 1-place functions goes back to the logicians Schönfinkel and Curry).

(82) ⟦*parent*⟧, reduced to 1-place functions:
λx Person[λy Person[y is a parent of x]]

This allows for statements like the following:

λx Person[λy Person[y is a parent of x]](Isaac)(Abraham)
= λy Person[y is a parent of Isaac](Abraham)
= 1, as Abraham is a parent of Isaac

Of course, we can apply the same technique for three-place relations as well:

(83) a. ⟦*give*⟧ as relation: { x, y, z | x gives y to z}
b. ⟦*give*⟧ as function: λ x,y,z [x gives y to z]
c. ⟦*give*⟧ as reduced function: λz[λy[λx[x gives y to z]]]

There are of course a number of ways to reduce a function. For example, instead of (81.b) we have the following, which differs only in the order in which the arguments are expected.

(84) ⟦*parent*⟧, reduced differently:
λy Person[λx Person[y is a parent of x]]

There is one important advantage of using 1-place functions in semantic analysis. Linguistic expressions that have two or more arguments often treat them differently, one being "closer" than the other. For example, the syntactic construction of the noun *father* shows that the child-argument is closer than the father-argument. The child argument forms a syntactic constituent with the noun, a so-called Noun Phrase (NP).

(85) a.Abraham is [$_{NP}$ Isaac's father].
b.Abraham is [$_{NP}$ the father of Isaac].

Also, the object argument of *love* is closer than the subject argument — the object forms a syntactic constituent with the verb, the so-called Verb Phrase (VP).

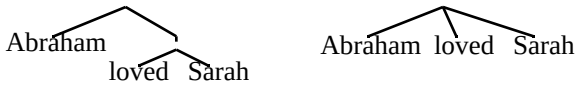
(86) Abraham [$_{VP}$ loved Sarah].

For the three-argument verb *give* things are a bit tricky. It is clear that the subject argument is the most removed. But the alternating forms in the following example suggest that *give* occurs in two frames that differ as to whether the object is the closest, or the recipient.

(87) a.Eve [$_{VP}$ gave an apple to Adam].
b.Eve [$_{VP}$ gave Adam an apple].

In general, we find syntactic evidence that favors binary-branching structures instead of ternary-branching structures:

(88)



And this favors the use of functions that take a simple argument over the use of more-than-one place relations, as we will see in the next section.

2.5. Conclusion

In this section we have discussed certain notions and tools that will be important for the analysis of the meaning of natural-language expressions, in particular sets, relations, and functions. We have seen that functions can be rendered in a particularly clear way in the lambda notation. We also found that sets can be reconstructed as functions to truth values, and that relations can be given as functions. This allows us to structure meanings in a way that accomodates the syntax of human languages better than the view of relations as sets of ordered tuples.