IEML Grammar

Pierre Lévy

January 17, 2014

The english version of this book has been reviewed and partially translated by Michele Healy, PhD

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

1.1 What IEML is about

1.1.1 A language endowed with computable semantics

If one wishes to augment human intelligence, it is better to possess a scientific theory of that intelligence. Now, human intelligence is intimately tied to the capacity to produce and to understand linguistic expressions, and to the capacity for symbolic manipulation in general. This is why a scientific model of human intelligence cannot exist without a scientific model of language, and in particular of the *semantic* dimension of language. The prime reason why I have designed IEML (Information Economy MetaLanguage) is to make available a language with a computable semantics – and not only a computable syntax.

1.1.2 A semantic coordinate system

Because of the interconnection of the transportation systems and the globalization of trade we saw in previous centuries a unification of the systems of weights and measures and the systems of geographic and temporal coordinates. Because of the general interconnection of information, we need now a system of semantic coordinates. I have designed IEML as a semantic coordinate system that is intended...

- to be used as a pivot language between natural languages (IEML automatically translates itself into natural languages),
- to improve the individual and collective management of knowledge
- to provide a reflective mirror to those collective intelligences that organize themselves in the digital medium,
- to promote an epistemological mutation in the humanities that would enable them to take advantage of the abundance of digital data.

1.1.3 A multi-purpose tool

IEML could be the basis for the simulation of ideas ecosystems, it fosters new methods for the analysis of massive flows of data ("big data"), it opens the way to previously unseen forms of translinguistic hypertextual communication, it supports an open and universal encyclopedic library that reorganize automatically according to the interests of its users, and it enables creative conversations to observe and perfect their own collective intelligence...

1.2 Responses to some objections

Let us specify the meaning of the terms: **IEML** is a formal language, and the **semantic sphere** is the system of coordinates generated by this language. The semantic sphere includes the set of all texts in IEML. In a previous book, entitled *The Semantic Sphere*[1] I contrast the research programs on augmented intelligence and artificial intelligence[2] and I show that the project based on IEML responds rather to the research program on augmented intelligence than to the research program on artificial intelligence, without rejecting - quite to the contrary - the acquisitions made by artificial intelligence. I also develop in this book the linguistic and semiotic arguments in favour of a universal computable metalanguage[3], because I am aware of the powerful prejudices according to which a universal language, let alone a calculable semantics, are "impossible" ... because they do not yet exist. Aviation, or even aircraft heavier than air, were deemed "impossible" for centuries before becoming an ordinary reality. The fact that each natural language frames reality in its own way, or that meaning depends on the context, are the two most common objections I receive. Yet these are not valid objections. First, IEML makes it possible to frame reality as one wishes. Second, it makes it possible to model contexts by means of hypertextual graphs, by variables representing value and intensity, by multimedia data and so on. In *The Semantic Sphere*, I also respond to those who fear that my system might limit the freedom of interpretations and conceptual creativity. In particular, I show that a digital medium equipped with the IEML semantic sphere will accommodate a hermeneutic memory that not only is conducive to semantic interoperability but also is a power for conceptual differentiation and interpretive diversification without precedent in the history of intellectual technologies[4].

[1] *The Semantic Sphere. Computation, cognition and the information economy.* ISTE Wiley, New-York and London, 2011.
[2] In Chapter 8.
[3] Notably in Chapters 10 and 11.
[4] In Chapters 13 and 14.

1.3 General plan of this book

In this book, I demonstrate step-by-step how to calculate semantic networks in natural language using algebraic expressions from IEML.

The chapter 2 on ***Syntax*** begins by presenting the regular language on which IEML is based, then shows that there is an automatic transformation between this regular language and a hyper-complex fractal graph called the IEML Rhizome.

Next, the chapter 3 on ***Semantics*** begins by presenting the structure of the IEML *dictionary*: a set of paradigmatic graphs that represent the semantic relations between the *terms*, and then the structure of the *statements* in IEML: phrases, texts and hypertexts.

The chapter 4 on ***Syntax-semantics transcoding*** contains one of the key elements of the system of the semantic sphere, because it formalizes the algorithms that transform the IEML syntax into semantics. In other words, this chapter explains step-by-step the transformation of a mathematical object (the regular language IEML) into a linguistic object: a set of semantic networks expressed in natural language.

Finally, the chapter 5 on ***Mathematical formalization*** presents the syntactic and semantic aspects of IEML in a mathematical mode. I show that the hypercomplex graph that underlies the internal semantic relations of the IEML semantic sphere (the Rhizome) is a group of calculable transformations (a "category", in the mathematical sense). The final section 5.7 of this last chapter suggests some quantitative methods (based on the spectral theory of graphs) that could be useful for measuring the structural similarities and distances within the IEML semantic sphere.

In order to fully grasp the content of this third volume on IEML grammar, the reader is invited to frequently refer to the figure 1.1 , which I will now discuss. The grey rectangles are *levels* and the spaces between these rectangles are *transcoders*.

1.4 The Algebra

The basic level at the lowest part of the figure 1.1 represents a relatively simple algebraic structure that I will call the IEML Algebra or, in abbreviated fashion, the *Algebra* (with a capital A). This is a group of transformations the variables of which can be calculated from the alphabet {E, U, A, S, B, T} using an additive operation and a multiplicative operation. This Algebra is described familiarly in the section 2.1. In the section 5.2 of the chapter 5 it is formalized as a regular language; the section 5.3 examines its group structure and the section 5.4 proves the calculability of the operations.

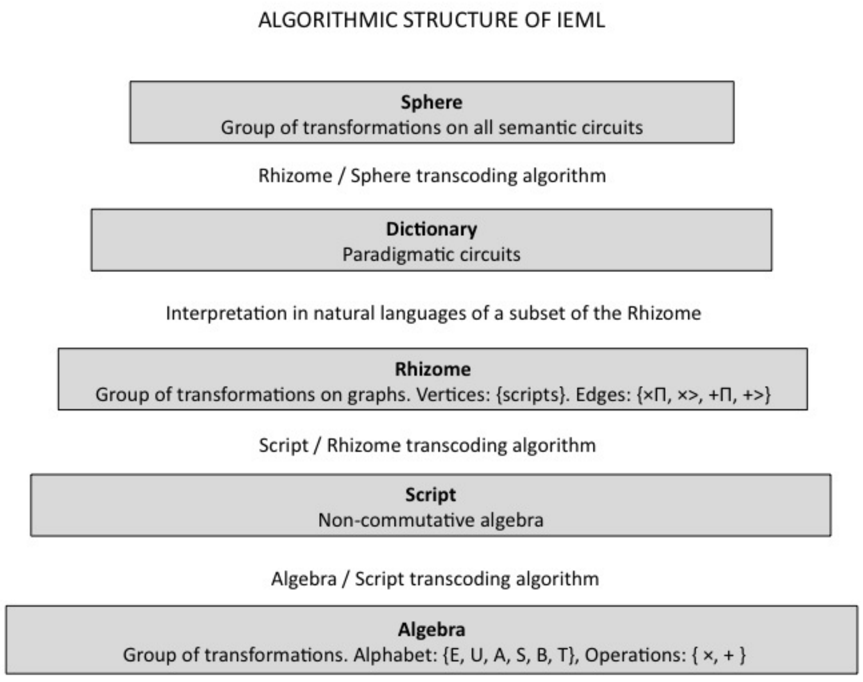

ALGORITHMIC STRUCTURE OF IEML
Sphere
Group of transformations on all semantic circuits
Rhizome / Sphere transcoding algorithm
Dictionary
Paradigmatic circuits
Interpretation in natural languages of a subset of the Rhizome
Rhizome
Group of transformations on graphs. Vertices: {scripts}. Edges: {×Π, ×>, +Π, +>}
Script / Rhizome transcoding algorithm
Script
Non-commutative algebra
Algebra / Script transcoding algorithm
Algebra
Group of transformations. Alphabet: {E, U, A, S, B, T}, Operations: { ×, + }

Figure 1.1: The general structure of IEML

1.5 The Script

The IEML Script or, in abbreviated fashion, the *Script*, is represented by the second level from the bottom of the figure 1.1. The Script is a particular notation of the Algebra, a notation that is described in the section 2.2. The various algebraic expressions that are *equivalent* are translated by a unique code in Script. Inversely, to each Script code corresponds one and only one valid algebraic expression.

Between the Algebra level and the Script level, a reciprocal transcoding algorithm operates, translating the algebraic expressions into Script and the Script codes into algebraic expressions. The structure of the non-trivial algorithm that codes the Algebra into Script is described informally in the section 2.3 and formalized in the section 4.2.

1.6 The Rhizome

The primary characteristic of the Script is to program the construction of the hyper-complex graph structure that is the Rhizome. The Rhizome is illustrated by the third level from the bottom in the figure 1.1. The vertices of the Rhizome are script codes and its edges represent the operation-based relations between the Script codes. The four main operations arise from algebraic operations: order ($\otimes >$) and multiplicative symmetry ($\otimes\sqcap$) , and order ($\oplus >$) and additive symmetry ($\oplus\sqcap$). The topology of the Rhizome is described in the section 2.4 and formalized in the sections 5.5 and 5.6. *This topology was conceived in order to support the semantic (paradigmatic, syntagmatic, textual and inter-textual) networks that organize the IEML units of meaning.*

Between the level of the Script and the level of the Rhizome a transcoding algorithm operates to translate the script codes into rhizomes and the rhizomes into script codes. The structure of this algorithm is described in the section 2.3 and formalized in the section 4.3.

1.7 The Dictionary

The chapter 2 on syntax involves only algebra, codes, algorithms and graphs, without any recognizable element of linguistics or semantics. But the first three levels from the bottom of the figure 1.1 represent the indispensable basis of a calculable semantics. The Script establishes an automatic correspondence between the IEML algebra - on the one hand - and the hyper-complex graphs of the Rhizome on the other. Thank to this correspondence the rhizomatic graphs are also the variables of a group of transformations (more exactly, a "category" in the mathematical sense) and that identity authorizes the *calculation of the hypercomplex graphs that support the semantic relations between the expressions of the IEML language.*

The human intervention begins at the third inter-level transcoder from the bottom of the figure 1.1. At this stage, semantic engineers choose subsets of the Rhizome and interpret them in natural languages. The result forms a dictionary, which is presented as a set of paradigmatic circuits. The methods of construction and the structure of the paradigmatic circuits of the dictionary are described in the section 3.1.

1.8 The circuits of the semantic sphere

Starting from the terms of the dictionary and the rules of grammar described in the section 3.2, it is finally possible to automatically construct and analyze circuits that are enunciative, in other words that are syntagmatic (propositions), textual, and hypertextual. The section 3.3 sketches the structure of the algorithm that is capable of constructing, from the Script and the Dictionary, the set of the hypercomplex circuits that explain in natural languages the semantic relations between the statements of the IEML language. This algorithm is formalized in the section 4.4.

1.9 Summary

1. There exists an automatic correspondence between, on the one hand, the semantic circuits proper to the IEML artificial language and, on the other hand, the IEML algebra, which by definition is calculable. *In other words, there exists an automatic correspondence between the syntax and the semantics of IEML.*

2. The paradigmatic circuits of the dictionary establish an automatable correspondence between the IEML artificial language and natural languages.

1.10 Acknowledgements

The research that led to IEML has been funded by the Canada Research Chair federal program and by the Social Sciences and Humanities Research Council of Canada.

Chapter 2

Syntax

«The unconscious is structured like a language» Lacan

As the reader already knows, the recognition, production, and transformation of discursive concepts in IEML is automatable, and it is automatable because it is mathematized. In this chapter we will examine, in the least technical manner possible, the mathematical underpinnings of the syntax of IEML and two of its transcoding algorithms. Please note that the algorithms will be formalized in the chapter 4, and the mathematics (or at the very least their foundations) in the final chapter (5) of this work. Also note that the chapter 3, which follows the present chapter, will address the linguistic and semantic aspects of the metalanguage, with translated examples in natural language.

2.1 The Algebra

2.1.1 The elementary symbols

The six elementary symbols {E, U, A, S, B, T} of our IEML Algebra correspond to the names of Empty (E), Virtual (U), Actual (A), Sign (S), Being (B) and Thing (T).

2.1.2 The sequences

2.1.2.1 How sequences are constructed

Now let us consider an algorithm capable of recognizing and manipulating the elementary symbols. One of its tasks consists of building semantic sequences by concatenating symbols. A sequence can be understood as an object fabricated by means of symbols that are aligned one behind the other, then bonded.

The length L of a sequence is equal to the number of symbols of which it is composed. These sequences are oriented, that is to say that they have a beginning and an end, or - if you will - an initial symbol and a terminal symbol.

L0	L1	L2	L3	L4	L5	L6
1	3	9	27	81	243	729

Table 2.1: Correspondance between layers (Ln) and length of the sequences

2.1.2.2 Layers of sequences

Sequences can have only 7 predefined lengths: 1, 3, 9, 27, 81, 243, 729. We see that, first, one symbol can be considered as a sequence of length L = 1 and that, second, the length of a sequence is always a power of 3 (3^0, 3^1, 3^2, 3^3, 3^4, 3^5, 3^6). We shall label a length of sequence, or *layer*, by the power of 3 that defines it. For example, we can speak of a sequence of *layer* L0, L1, L2, etc., up to L6[1].

To construct a sequence of layer L0, it suffices for the Algebra to take one single symbol. To construct a sequence of layer L1, the Algebra attaches three sequences of length L0, end-to-end. To construct a sequence of layer L2, it attaches three sequences of layer L1, end-to-end, and so on. Our Algebra cannot construct a sequence of layer Ln (where n is greater than zero and less than or equal to 6) if it has not previously constructed the three sequences of layer Ln–1 that it requires to do so.

As a general rule, to construct a sequence of layer Ln (where n is greater than zero and less than or equal to 6), the Algebra attaches, end-to-end, three sequences of layer Ln–1.

2.1.2.3 The three semes of a sequence: substance, attribute, mode

The three sequences of layer Ln–1 that serve to construct a sequence of layer Ln are always identifiable in the Algebra. Moreover, because the sequences have a beginning and an end, an algebraic algorithm can determine which of the three sub-sequences is the first, which is the second, and which is the third. These three sub-sequences are called semes. The first seme is called the substance of the sequence, the second is called the attribute and the third is called the mode. For the grammatical significance of semes, see 3.2.3.

2.1.2.4 Multiplication

The algebraic operations to construct sequences are called *multiplications*. Here, the term "multiplication" is clearly taken in a different sense than the multiplication of numbers. Here it is semes (i.e., semantic sequences) that are the operands of the multiplication. If an algorithm is capable of constructing sequences, then an algorithm also can deconstruct them, for example by decomposing a sequence into three semes of the immediately lower layer. An algorithm also can transform the sequences that it has constructed, for example by replacing one seme by another.

[1]L corresponds to the first letter of the English word *layer*.

2.1.3 Sets of sequences

2.1.3.1 Types of sets

We have just seen that our algebraic algorithms are capable of constructing, recognizing, decomposing and transforming sequences. Now we will see that they also can pack, recognize, unpack and transform *packets* of sequences, packets of packets, and so on. A packet of sequences manipulated by an algebraic algorithm cannot contain identical sequences. Likewise, a packet of sub-packets cannot contain identical sub-packets. Each sequence in a sub-packet and each sub-packet in a packet is unique. This is why we call these packets *sets*. The three main sorts of sets manipulated by our algorithms from the algebraic level are as follows:

- the *categories*, which are sets of sequences from the same layer;
- the *catsets*, which are sets of categories from the same layer;
- the *USL*s, which are sets of categories from different layers, and which therefore can always be organized in sets of 1 to 7 catsets from distinct layers.

The term *set of sequences* is a very general expression that can designate a sequence, a category, a catset, a USL, a set of USLs, etc.

2.1.3.2 Addition

All the operations that manipulate sets of sequences, such as union, intersection or symmetrical difference, are called *additive operations*, because the result of these operations can always be understood as a union of sets of sequences. As in the case of semantic multiplication, semantic addition clearly is taken in a different sense than the addition of numbers.

2.1.4 Coding of the Algebra

The operations and variables of our system of sequences and of sets of sequences are integrally defined, and the sets of sequences are finite in number. The operations on variables can be described by algorithms. In other terms, a circuit of algorithms from the level of the Algebra is capable of constructing, deconstructing and transforming sequences or sets of sequences. In the same way, the circuit of an algorithm can describe the sequences that it constructed and the sets of sequences that it enveloped.

We can imagine a very large number of different notations for the IEML Algebra. Yet all these notations are functionally equivalent from the point of view of the results obtained by the algorithms that can decode these notations. The notation that I typically adopt is quite close to the "reality" being described (the system of sequences). I mark the multiplicative operations by a pure and simple concatenation of symbols: for example {EUT} denotes a sequence from

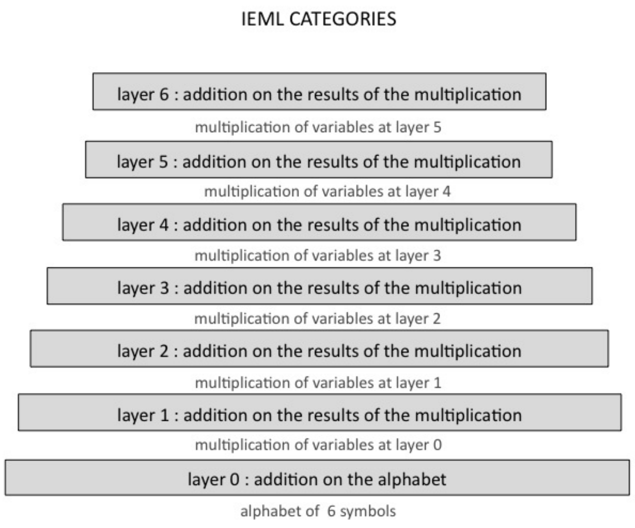

IEML CATEGORIES
layer 6 : addition on the results of the multiplication
multiplication of variables at layer 5
layer 5 : addition on the results of the multiplication
multiplication of variables at layer 4
layer 4 : addition on the results of the multiplication
multiplication of variables at layer 3
layer 3 : addition on the results of the multiplication
multiplication of variables at layer 2
layer 2 : addition on the results of the multiplication
multiplication of variables at layer 1
layer 1 : addition on the results of the multiplication
multiplication of variables at layer 0
layer 0 : addition on the alphabet
alphabet of 6 symbols

Figure 2.1: The layers of categories in IEML

layer L1 in which E is the substance, U the attribute and T the mode. I mark the additive operations by an enumeration that uses the comma. For example {EUT, UAS, TUE} represents a category from layer L1 containing 3 sequences.

2.1.4.1 Mathematical characterization, calculability

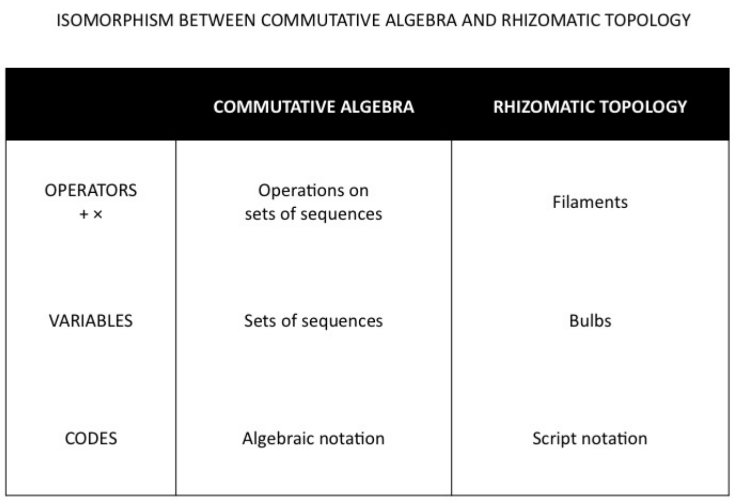
At this stage, IEML refers to a trivial commutative algebra: the cascading Cartesian multiplications that generate its sequences (the multiplicative operations) and the operations that assemble the sub-sets of sequences (the additive operations) can be traced back to the most classical logical functions. Seen from the angle of formal languages, the IEML Algebra refers to well-repertoried operations that can be conducted on the expressions of a finite regular language. From the functional point of view, it is a category (a complex group) of transformations. Remember that the word "category" has two distinct meanings here: (a) the meaning as in mathematics (a complex group of transformations) and (b) the meaning that it has in the technical vocabulary of IEML, namely a set of sequences from the same layer: see the figure 2.1.

2.1.4.2 Several notations exist for a given result

I would like to stress the commutativity of operations in the IEML Algebra. Its additive operations obviously are commutative.

For example, the result of :

{{E, U, S}, {S, B, E}}



ISOMORPHISM BETWEEN COMMUTATIVE ALGEBRA AND RHIZOMATIC TOPOLOGY

	COMMUTATIVE ALGEBRA	RHIZOMATIC TOPOLOGY
OPERATORS + ×	Operations on sets of sequences	Filaments
VARIABLES	Sets of sequences	Bulbs
CODES	Algebraic notation	Script notation

Figure 2.2: Correspondence between Algebra and Rhizome

is the same at that of :
{{E, S, B}, {E, U, S}},
namely:
{E, U, S, B}.
We also see that the associativity of the additive and multiplicative operations makes it possible to arrive at the same result in several ways.
For example,
{{S{B, A}T}, {SET}}
gives the same result as:
{S{E, A, B}T},
namely:
{SET, SAT, SBT}.
Thus, even if we stay within the limits of a single system of algebraic notation, there are nonetheless various ways of coding the same set of sequences.

2.2 The Script

2.2.1 Conventions of writing: Script and script

I will write *Script* (with a capital S) to designate **the code** for our system of rhizomatic topology, and *script* (with a small s) to designate **a particular**

expression of that code. The notation Script replaces the notation STAR that was referred to in *The Semantic Sphere*.

2.2.2 Biunivocal correspondence between sets of sequences and scripts

We have just seen that several possible systems of notation exist in the IEML Algebra. The Script is one such system. It is a particular way of noting the Algebra. This means that a notation in Script is *ipso facto* a notation in IEML Algebra. As a result, ascript designates one and only one set of sequences. The passage from IEML Script to the IEML Algebra is thus ensured. This passage poses no problems, because the first is a particular case of the second. On the other hand, the inverse passage, from the Algebra to the Script, is more complex. This is why we have a *transcoding algorithm* [2] that transforms the expressions from the Algebra into scripts. This algorithm respects the constraint according to which there always exists one and only one script to describe a set of sequences. So, in summary:

- all the different expressions of the commutative Algebra that designate the same set of sequences are translated by one and only one script;
- a distinct script designates one and only one set of sequences and it always has a translation into commutative Algebra.

This biunivocal correspondence between the two codes has important consequences. Indeed, the variables, operations and code of a formal system act in concert. If we can move automatically from the code of a formal system A to the code of a formal system B and *vice versa*, then we also can move automatically - via the intermediary of the code - from the variables and operations of system A to the variables and operations of system B (and *vice versa*).

2.2.3 Biunivocal correspondence between scripts and rhizomes

A script encodes not only a *set of sequences*, but also must encode simultaneously, and in a non-ambiguous manner, a complex *graph* called a rhizome (see below: 2.4). In other words:

- each distinct IEML script is interpreted by an algorithm as the description of one and only one distinct rhizome (see below in2.5 how this algorithm functions);
- a distinct rhizome is described or recognized by an algorithm in the form of one and only one distinct script.

[2] See below, 2.3 and 4.2

2.2.4 Biunivocal correspondence between sets of sequences and rhizomes

There exists a biunivocal correspondence between sets of sequences and scripts, on the one hand, and between scripts and rhizomes, on the other hand. Consequently, there necessarily exists a biunivocal correspondence between sets of sequences and rhizomes.

2.2.5 Solidarity between the operations of algebraic calculation and those of rhizomatic calculation

Because of the biunivocal correspondence between sets of sequences and rhizomes, there also is a mechanical solidarity between the algebraic calculation that constructs, transforms, and describes the sets of sequences, on the one hand, and the rhizomatic calculation that constructs, transforms, and describes the rhizomes, on the other hand.

2.2.6 Rules of notation in Script

2.2.6.1 Operators

Multiplication : . - ' , _ ; Instead of being marked by a simple concatenation of symbols, multiplication is indicated in Script by *punctuation marks* indicating the different layers. These multiplication signs facilitate the automatic syntactic analysis (*parsing*) of the expressions and make it possible to visually detect the semes (that is to say the operands of the multiplication), layer by layer. The multiplication signs in Script are as follows:

L0	:
L1	.
L2	-
L3	'
L4	,
L5	_
L6	;

For example, the sequence:
STAUBSTAUESSUBSTAUEBSTBSAAU
is punctuated this way in Script:
S:T:A:.U:B:S:.T:A:U:.-E:S:S:.U:B:S:.T:A:U:.-E:B:S:.T:B:S:.A:A:U:.-'

Algebra	Script	Signification
{U, A}	O:	verb, process
{S, B, T}	M:	noun, entity
{U, A, S, B, T}	F:	fullness (absence of emptiness)
{E, U, A, S, B, T}	I:	information

Table 2.2: Abbreviations of additions from layer L0

Addition (+) Instead of being marked by an enumeration between braces, addition is indicated in Script by the sign + as well as by an opening parenthesis and a closing parenthesis.

For example, {UAS, AUT, TUA} is punctuated as (U:A:S:. + A:U:T:. + T:U:A:.) in Script.

2.2.6.2 Delimiters

Delimiters of categories Distinct categories are separated by a (forward) slash “/”. For example :

(U:A:S:. + A:U:T:. + T:U:A:.) / S:T:A:.U:B:S:.T:A:U:.-E:S:S:.U:B:S:.T:A:U:.-E:B:S:.T:B:S:.A:A:U:.-’

Delimiters of USL * ** The reader will recall that USLs are sets of distinct categories. If it is necessary to delimit a USL in Script, then it is preceded by one asterisk "*" and followed by two asterisks "**".

For example:

*(U:A:S:. + A:U:T:. + T:U:A:.) / S:T:A:.U:B:S:.T:A:U:.-E:S:S:.U:B:S:.T:A:U:.-E:B:S:.T:B:S:.A:A:U:.-’**

2.2.6.3 Mandatory abbreviations

Abbreviation of additions from layer L0 The abbreviation of the four additions from layer 0 as seen in the table 2.2 is mandatory.

Abbreviation of multiplications of E: The use of abbreviation for the multiplications of E: is mandatory. When an E: is multiplied by other E: (plural) until the end of a category, it will suffice to mark the layers without filling them with E: (plural).

I use the pound sign (#) before incorrect expressions in Script.

<u>Example 1</u>

We do not write #E:E:E:. but rather *E:.**

<u>Example 2</u>

We do not write #O:E:E:.E:M:E:.E:E:E:.- but rather *O:.E:M:.E:.-**

<u>Example 3</u>

We do not write #U:E:E:.E:E:E:.E:E:E:.- but rather *U:.-**

The 25 lower-case letters in Script
(layer 1)

Attribute → / Substance ↓	U	A	S	B	T
U	UUE **wo.** reflect (internal)	UAE **wa.** move	USE **y.** know	UBE **o.** want	UTE **e.** can
A	AUE **wu.** percieve	AAE **we.** reflect (external)	ASE **u.** state	ABE **a.** commit	ATE **i.** do
S	SUE **j.** signifier mutation	SAE **g.** documentary mutation	SSE **s.** thought	SBE **b.** language	STE **t.** memory
B	BUE **h.** signified mutation	BAE **c.** personal mutation	BSE **k.** society	BBE **m.** affect	BTE **n.** world
T	TUE **p.** mutation of referent	TAE **x.** material mutation	TSE **d.** truth	TBE **f.** life	TTE **l.** space

Figure 2.3: 25 lower-case letters

Abbreviation of 25 sequences from layer 1 The use of the 25 lower-case letters that encode certain sequences from layer 1 is mandatory.

2.2.7 Standard order

The standard order of the Script provides regular procedures for the arrangement of categories, catsets, and USLs.

2.2.7.1 Ordering by layer

Sets of sequences are always arranged in layer order . For example, we rank the sets from layer 0 before the sets from layer 1, and the sets from layer 1 before the sets from layer 2, etc.

2.2.7.2 Ordering by size

Ordering by size is done according to increasing number of singular sequences by category, increasing number of distinct categories by catsets, etc. This means that the sets of sequences (belonging to the same type) are ranked by increasing number of elements.

For example:

E:	U:	A:	S:	B:	T:
1	2	4	8	16	32
2^0	2^1	2^2	2^3	2^4	2^5

Table 2.3: Numerical value of primitives

A:B:S:. contains only one single sequence {ABS}
O:B:S:. contains two sequences{UBS, ABS}
M:B:S:. contains three sequences {SBS, BBS, TBS}
O:M:S:. contains six sequences {USS, UBS, UTS, ASS, ABS, ATS}

2.2.7.3 Alphabetical order of primitives, characters, and lower-case letters

Each primitive symbol holds a numerical value (see the table 2.3). The numerical value of an addition of primitives results from the arithmetical addition of the numerical values of the operands. For example, the numerical value of (S:+B:) is 24 and the numerical value of O: is 6.

Each of the 63 additions from layer L0[3] is considered to be a *character* of the Script. The alphabetical order of the script arranges the characters according to their increasing numerical value.

In agreement with the principles that have just been described, the alphanumeric order of the 10 capital letters is:
E: U: A: O: S: B: T: M: F: I:
and that of the 25 lower-case letters is as follows (see the figure 2.3):
wo. wa. y. o. e. wu. we. u. a. i. j. g. s. b. t. h. c. k. m. n. p. x. d. f. l.

2.2.7.4 Order of categories from the same layer

The standard order of the Script arranges the categories from the same layer by increasing size, and then by alphabetical order.

Example 1

The categories from layer 1 shown here below are ranked according to the order of categories. The boldface numbers indicate the **size** of the categories.

1
E:B:T:.
A:B:E:.
A:B:S:.
3
E:B:M:.
M:S:E:.
5
U:F:E:.
6

[3]I.e., the set of all subsets of {E, U, A, S, B, T} minus the empty set.

E:S:I:.
U:M:O:.
U:I:E:.
A:T:I:.
9
U:M:M:.
30
I:E:F:.
180
I:F:I:.
Example 2

Arrangement by alphabetical order of four categories from layer L2 that have the same size:
O:O:.M:M:.-
O:M:.O:M:.-
O:M:.M:O:.-
M:M:.O:O:.-

2.2.7.5 Order of the operands of an addition

The operands of additions are always arranged in standard order.
Example 1
We never write #(S:+E:+U:) but always *(E:+U:+S:)**
Example 2
We never write #(M:+U:) but always *(U:+M:)**

2.2.7.6 Order of catsets

The catsets are in principle already arranged "internally" according to the order of the categories.

1. A set of catsets is first arranged by layers.
2. Catsets from the same layer are arranged in increasing order of size (that is to say by number of categories).
3. Catsets of the same size and from the same layer are arranged in alphabetical order *from their first category.* If the first categories are identical, they are arranged by alphabetical order from their second category and so forth until they all are ordered.

2.2.7.7 Order of USLs

In principle, USLs are already arranged into catsets from distinct layers (from 1 to 7 catsets), with the catsets themselves being arranged in order of categories. USLs are ranked by increasing layers, in the order of their catsets. That is to say that the USLs that begin with a catset from layer 0 come first, then the USLs that begin with a catset from layer 1, and so on.

2.2.8 Writing constraints in Script

It is understood that scripts can always be interpreted as instructions for additive and multiplicative operations on sequences. But the scripts must also be able to be interpreted as instructions for «topological» operations of construction or identification of rhizomes. Note that, to meet this second constraint, the Script is presented as a non-commutative and non-associative algebra. There exists one and only one script that can encode a distinct rhizome.

2.2.8.1 Addition in Script is non-commutative and non-associative

Non-commutativity The operands of additions between categories from the same layer are arranged in the standard order. This implies the absence of commutativity of the addition.

Example
We *cannot* write that:
*(U:+A:)** is identical to#(A:+U:)
because we cannot write:
#(A:+U:)

Non-associativity In Script, it is impossible to write *several* different additions between categories from layer Ln within one seme (operand of multiplication) from layer Ln. Nor can we write an addition of additions.

Example
In Script, in is incorrect to write:
#(y.+ a.)+(F:E:O:.+O:F:E:.)
Rather, we must write:
*(y. + a.+ O:F:E:.+F:E:O:.)**

2.2.8.2 Minimization of the number of multiplications in Script

In Script we cannot write an addition of several multiplications from layer Ln if it is possible to write the same result by a single multiplication of additions from layer Ln–1. In other words, we always use the smallest possible number of multiplications from layer Ln. This implies an absence of associativity in the composition of additions and multiplications in Script.

Example 1
We cannot write:
#(U:S:A:. + U:S:B:. + U:T:A:. + U:T:B:.)
but rather we must write:
*U:(S:+T:)(A:+B:).**
We see that the incorrect expression contains four multiplications from layer L1 whereas the correct expression contains only one. The correct expression minimizes the number of multiplications.

Example 2
Consider the category {USE, UBE, UTE, ASE, ABE, ATE} in commutative Algebra.

This category cannot be represented in Script by:
#((y.+ o. + e.) + (A:M:.))
nor by:
#(U:M:. + A:M:.)
In Script, the sequence can only be represented by:
*O:M:.**

2.2.8.3 Choice between several scripts offering the same minimal number of multilpications

When there exist several scripts that offer the same minimal number of multiplications, we systematically take the one that is situated first in the standard order.

Example

Consider the category {USS, UTS, UTB, STB, SSB, SSS} noted in commutative Algebra.

There are no scripts that diminish the number of multiplications beyond three. On the other hand, there do exist two scripts that offer a translation of the algebraic expression with only three multiplications. Thus, between these two scripts, we must choose the one that comes first in the standard order. The other is incorrect.

*(U:(S:+T:)S:. + S:S:(S:+B:). + (U:+S:)T:B:.)**
#(U:T:(S:+B:). + S:(S:+T:)B:. + (U:+S:)S:S:.)

2.3 Principles of the transcoding algorithm from the Algebra into the Script

2.3.1 General architecture of the coding algorithm in Script for a category from layer Ln

The replacement operation is explained in the following section.

1. Replacement operation (recursive) for the category of layer Ln
2. Replacement operation (recursive) for the categories of layer Ln–1
3. etc.
4. Replacement operation (recursive) for the categories of layer L1

2.3.2 Replacement operation on a category of layer Lk

Any time that n (the largest possible n) sequences belonging to the same IEML category of layer L have two similar semes (for example, the substance and the mode), they are replaced by a single sequence, i.e., by a sole multiplication. This replacement is done by application of the following two rules:

1. Always seek to minimize the number of multiplications that result in a category of layer Ln.
2. When several categories are available to obtain the same number of minimal multiplications, systematically take the category situated first in the standard order as described in 2.2.7.

2.3.3 Reiteration of the replacement operation

The replacement operation is repeated on the result of the first operation, in a loop, until no more sequences of characters with two similar semes can be found.

2.3.4 Examples of the coding algorithm in Script

Example 1 (layer L1) : {USE, UBE, UTE, ASE, ABE, ATE}

- USE, UBE, UTE have 2 similar semes, and thus can be represented by: U:(S:+B:+T:)E:.
- ASE, ABE, ATE have 2 similar semes, and thus can be represented by: A:(S:+B:+T:)E:.
- Thus we obtain: #U:(S:+B:+T:)E:.+A:(S:+B:+T:)E:.
- The two operands of the union obtained have two semes in common. Thus, we represent the union as: #(U:+A:)(S:+B:+T:)E:.
- In order to obey the constraints of abbreviated notation, the last result is represented as: *O:M:.**

Example 2 (layer L1) : {AUB, ABE, SUE, SBE, TUE, TBE}

- ABE, SBE and TBE can be represented by: (A:+S:+T:)B:E:.
- SUE and TUE can be represented by (S:+T:)U:E:.
- In this way we obtain: *(A:U:B:.+(S:+T:)U:E:.+(A:+S:+T:)B:E:.)** Note that the standard order requires arranging the operands of addition in order of increasing size. The substance of the first operand is of size 2 while the substance of the second operand is of size 3.

Example 3 (layer L2) : {AUBABESUE, AUBABEABE}

- {AUBABESUE, AUBABEABE} is written *A:U:B:.A:B:E:.(A:B:E:.+S:U:E:.)-**

FILAMENTS	**Multiplication** $\otimes$	**Addition** $\oplus$
Order $>$	$\otimes >$	$\oplus >$
Symmetry $\sqcap$	$\otimes\sqcap$	$\oplus\sqcap$

Table 2.4: The 4 types of filaments of an IEML rhizome

Example 4 (layer L2) : {USEUBEUTE, ASEABEATE, ASEABEUTE}

- Replacement for layer 2
  #(U:S:E:.U:B:E:.U:T:E:.- + A:S:E:.A:B:E:.(U:T:E:.+A:T:E:.)-)
- Replacement for layer 1
  *(U:S:E:.U:B:E:.U:T:E:.- + A:S:E:.A:B:E:.(U:+A:)T:E:.-)**

2.4 The Rhizome

2.4.1 Bulbs and filaments

Here, the term rhizome is used in a sense that has only a distant connection to the world of botany, referring instead to the philosophical meaning of a *regular fractal graph that is more complex than a classic tree structure*[4]. An IEML rhizome is fundamentally a graph in which the vertices, called *bulbs*, are labelled by a script and in which the edges, called *filaments*, are labelled according to the relations that they establish between the bulbs. Because only four types of relations exist between the bulbs, there are only four labels of filaments[5]. The relations between the bulbs arise directly from the two operations (addition and multiplication) and are named as follows:

1. Additive order
2. Additive symmetry
3. Multiplicative order
4. Multiplicative symmetry

From the point of view of graph theory, order relations construct trees, whereas symmetrical relations construct cliques.

In additive trees, the number of leaves a root may have is not specified. The leaves and roots of additive trees are bulbs of the same layer.

In multiplicative trees, each root can have only three leaves, corresponding to the three operands of the multiplication. The leaves of multiplicative trees always belong to the layer immediately below the layer of their roots.

[4]See Deleuze and Guattari, *A Thousand Plateaus*, trad. Brian Massumi, Continuum, NY and London, 2004 (original version: Minuit, Paris, 1980), in particular the introduction.

[5]In the section 3.1.6 we will see that there are 4 more, for a total of 8, but we can ignore this for the time being.

2.4.2 Notation of rhizomes and definition of the four types of filament

Given the special constraints imposed on the Script (to establish a biunivocal relationship between the Algebra and the Rhizome), a *detailed description of the rhizomes cannot be expressed in Script.* Therefore I must call upon a special "topological" notation , which is distinct from the notation of Algebra and that of the Script. In principle, this notation is used *only* to describe the structure of the Rhizome.

2.4.2.1 Bulbs

A bulb (in other words an vertex of the rhizomatic graph) is marked here by a small circle before the script that it labels (without stars).

For example: °M:S:U:.

2.4.2.2 Filaments of additive order

The relation of additive order is marked by the sign $\oplus >$

The sign of additive order indicates the relation between the result (to the left of the sign) and the operand (to the right of the sign) of an addition. Recall that, within a category, the bulbs in a relation of additive order always belong to the same layer.

For example, °M:S:U:. $\oplus >$ °B:S:U:. indicates that °M:S:U:. additively *contains* the bulb °B:S:U:. In other words, the set of sequences labelling the bulb °B:S:U:. *is a sub-set of* the set of sequences labelling the bulb °M:S:U:.

2.4.2.3 Filaments of additive symmetry

A symmetrical additive relation is marked by the sign $\oplus\sqcap$ between brackets.

The sign of multiplicative symmetry indicates that the bulbs that it connects are operands of the same addition. Invariably there exists a *complete graph* of additive symmetrical relations between the operands of an addition, and these operands of course always belong to the same layer (within a category).

For example, [°S:S:U:. $\oplus\sqcap$ °B:S:U:. $\oplus\sqcap$ °T:S:U:.] indicates that each of the three bulbs is in a relation of additive symmetry with the other two, or that °S:S:U:. °B:S:U:. and °T:S:U:. are the operands of one and the same addition operation.

2.4.2.4 Compact notation of the additive relations that correspond to one single addition

Instead of successively indicating several relations of order as in:

°M:S:U:. $\oplus >$ °S:S:U:.
°M:S:U:. $\oplus >$ °B:S:U:.
°M:S:U:. $\oplus >$ °T:S:U:.

we can write in the following manner the set of relations of order and of additive symmetry that correspond to one single addition:
°M:S:U:. $\oplus >$ [°S:S:U:. $\oplus\sqcap$ °B:S:U:. $\oplus\sqcap$ °T:S:U:.]

2.4.2.5 Filaments of multiplicative order

A filament of multiplicative order is marked by the sign $\otimes >$

The sign of multiplicative order indicates the relation between the result (to the left of the sign) and the operand (to the right of the sign) of a multiplication. Recall that the operand always belongs to the layer immediately below that of the result.

For example:
°M:S:U:. $\otimes >$ substance °M:
°M:S:U:. $\otimes >$ attribute°S:
°M:S:U:. $\otimes >$ mode °U:

Instead of successively indicating the three relations of multiplicative order, we can use the following notation, in which the order of the semes indicates their distinct roles:
°M:S:U:. $\otimes >$ [°M: $\otimes$ °S: $\otimes$ °U:]

2.4.2.6 Filaments of multiplicative symmetry

A filament of multiplicative symmetry is marked by the sign $\otimes\sqcap$

The sign of multiplicative symmetry indicates that the two bulbs of layer n that it connects have two of their semes of layer n-1 in an inversed role, while the third seme of layer n-1 is identical. The bulbs in relation of multiplicative symmetry necessarily belong to the same layer.

For example : °M:S:U:. $\otimes\sqcap$ °S:M:U:.

2.5 Principles of the algorithm transcoding the Script into the Rhizome

*A script is considered a program for the construction of rhizome*s. The automatic reading of the script ultimately leads back to the operations for identification of the bulbs and for the weaving of filaments between the identified bulbs. The word "character" as used below refers to *the 63 sub-sets* of {E, U, A, S, B, T} minus the empty set. Thus, not only the ten symbols {E, U, A, O, S, B, T, M, F, I} are involved.

2.5.1 General structure of the algorithm for construction of a rhizome, starting from a script of layer Ln

The construction of a rhizome starting from a script of layer Ln (n between 0 and 6) is broken down as follows:

1. Construction for the layer Ln,

2. Construction for the layer Ln–1,
3. ...
4. Construction for the layer L0

2.5.2 General structure of the routine for construction of a rhizome for one layer

1. First round:
   (a) Identification of the bulbs of layer Lk in a relation of addition, as delivered by the script.
   (b) Weaving of filaments of order and of additive symmetry from layer Lk between the bulbs identified in (a).
   (c) Weaving of filaments of multiplicative order (roots to the layer Lk–1) from the bulbs identified in (a).
2. Second round: successive construction, character by character, of the bulbs of layer Lk identified in the first round.
3. Third round: successive construction, character by character, of the bulbs of layer Lk identified in the second round.
4. ...
5. Second-last round: we continue in this fashion until all the bulbs identified correspond to singular sequences.
6. Last round: we weave the filaments of multiplicative symmetry between the bulbs identified in the preceding rounds.

2.5.3 Passage from one layer to the layer below it

Construction of each layer gives the multiplicative roots of the layer immediately below it, as we see in point 1 (c) of2.5.2. *These are the roots that serve as a basis for the construction of the rhizome at the lower layer, and so on recursive*ly.

2.5.4 General structure of the routine for construction of bulbs

1. Construction starting from the first character
   (a) Identification of the characters in additive relation *contained* in the first character, identification of the corresponding bulbs.
   (b) Weaving of the filaments of order and of additive symmetry from layer Lk between the bulbs identified in (a).

   (c) Weaving of the filaments of multiplicative order (roots to the layer Lk–1) of the bulbs identified in (a).

2. Construction starting from the second character

   (a) Identification of the characters in additive relation *contained* in the second character, identification of the corresponding bulbs.
   (b) Weaving of the filaments of order and of additive symmetry from layer Lk between the bulbs identified in (a).
   (c) Weaving of the filaments of multiplicative order (roots to the layer Lk–1) of the bulbs identified in (a).

3. ...

4. Construction starting from the last character.

2.5.5 Examples of construction of a rhizome

First round for a layer of *O:M:(U:+B:). + M:O:(A:+T:).**

1. Identification of the bulbs of layer L1 in additive relation given by the script.
   °O:M:(U:+B:). + M:O:(A:+T:).
   °O:M:(U:+B:).
   °M:O:(A:+T:).

2. Weaving of filaments of order and of additive symmetry from layer L1 between the bulbs identified in 1.
   °O:M:(U:+B:). + M:O:(A:+T:). ⊕ > [°O:M:(U:+B:). ⊕⊓ °M:O:(A:+T:).]

3. Weaving of the filaments of multiplicative order (roots to the layer L0) from the bulbs identified in 1.
   °O:M:(U:+B:). ⊗ > [°O: ⊗ °M: ⊗ °(U:+B:)]
   °M:O:(A:+T:). ⊗ > [°M: ⊗ °O: ⊗ °(A:+T:)]

Construction starting from the bulb °O:M:(U:+B:).

1. Construction starting from the first character:

   (a) Identification of the characters in additive relation *contained* in the first character.
   U: and A:
   identification of the corresponding bulbs.
   °U:M:(U:+B:). and °A:M:(U:+B:).
   (b) Weaving of the filaments of order and of additive symmetry from layer L1 between the bulbs identified in (a).
   °O:M:(U:+B:). ⊕ > [°U:M:(U:+B:). ⊕⊓ °A:M:(U:+B:).]

(c) Weaving of the filaments of multiplicative order (roots to the layer L0) from the bulbs identified in (a).
°U:M:(U:+B:). ⊗ > [°U: ⊗ °M: ⊗ °(U:+B:)]
°A:M:(U:+B:). ⊗ > [°A: ⊗ °M: ⊗ °(U:+B:)]

2. Construction starting from the second character:

   (a) Identification of the characters in additive relation *contained* in the second character.
   S: B: and T:
   Identification of the correspinding bulbs.
   °O:S:(U:+B:).
   °O:B:(U:+B:).
   °O:T:(U:+B:).

   (b) Weaving of the filaments of order and of additive symmetry from layer L1 between the bulbs identified in (a).
   °O:M:(U:+B:). ⊕ > [°O:S:(U:+B:). ⊕⊓ °O:B:(U:+B:). ⊕⊓ °O:T:(U:+B:).]

   (c) Weaving of the filaments of multiplicative order (roots to the layer L0) from the bulbs identified in (a).
   °O:S:(U:+B:). ⊗ > [°O: ⊗ °S: ⊗ °(U:+B:)]
   °O:B:(U:+B:). ⊗ > [°O: ⊗ °B: ⊗ °(U:+B:)]
   °O:T:(U:+B:). ⊗ > [°O: ⊗ °T: ⊗ °(U:+B:)]

3. Construction starting from the third character:

   (a) Identification of the characters in additive relation *contained* in the third character.
   U: and B:
   Identification of the corresponding bulbs.
   °O:M:U:.
   °O:M:B:.

   (b) Weaving of the filaments of order and of additive symmetry from layer L1 between the bulbs identified in (a).
   °O:M:(U:+B:). ⊕ > [°O:M:U:. ⊕⊓ °O:M:B:.]

   (c) Weaving of filaments of multiplicative order (roots to the layer L0) from the bulbs identified in (a).
   °O:M:U:. ⊗ > [°O: ⊗ °M: ⊗ °U:]
   °O:M:B:. ⊗ > [°O: ⊗ °M: ⊗ °B:]

<u>Construction starting from the first character of °I:O:.</u>

1. Orders and additive symmetries from layer 0 for the first character
   °I: ⊕ > [°E: ⊕⊓ °F:]
   °F: ⊕ > [°O: ⊕⊓ °M:]
   °O: ⊕ > [°U: ⊕⊓ °A:]
   °M: ⊕ > [°S: ⊕⊓ °B: ⊕⊓ °T:]

2. Corresponding bulbs from layer L1
   - (a) ˚E:O:. ⊗ > [˚E: ⊗ ˚O: ⊗ ˚E:]
   - (b) ˚U:O:. ⊗ > [˚U: ⊗ ˚O: ⊗ ˚E:]
   - (c) ˚A:O:. ⊗ > [˚A: ⊗ ˚O: ⊗ ˚E:]
   - (d) ˚O:O:. ⊗ > [˚O: ⊗ ˚O: ⊗ ˚E:]
   - (e) ˚S:O:. ⊗ > [˚S: ⊗ ˚O: ⊗ ˚E:]
   - (f) ˚B:O:. ⊗ > [˚B: ⊗ ˚O: ⊗ ˚E:]
   - (g) ˚T:O:. ⊗ > [˚T: ⊗ ˚O: ⊗ ˚E:]
   - (h) ˚M:O:. ⊗ > [˚M:⊗ ˚O: ⊗ ˚E:]
   - (i) ˚F:O:. ⊗ > [˚F: ⊗ ˚O: ⊗ ˚E:]
3. Weaving of filaments of order and of additive symmetry between the bulbs of layer L1
   - (a) ˚O:O:. ⊕ > [˚U:O:. ⊕⊓ ˚A:O:.]
   - (b) ˚M:O:. ⊕ > [˚S:O:. ⊕⊓ ˚B:O:. ⊕⊓ ˚T:O:.]
   - (c) ˚F:O:. ⊕ > [˚O:O:. ⊕⊓ ˚M:O:.]
   - (d) ˚I:O:. ⊕ > [˚E:O:. ⊕⊓ ˚F:O:.]
4. Weaving of filaments of multiplicative order (roots to layer L0)
   - (a) ˚E:O:. ⊗ > [˚E: ⊗ ˚O: ⊗ ˚E:]
   - (b) ˚U:O:. ⊗ > [˚U: ⊗ ˚O: ⊗ ˚E:]
   - (c) ˚A:O:. ⊗ > [˚A: ⊗ ˚O: ⊗ ˚E:]
   - (d) ˚O:O:. ⊗ > [˚O: ⊗ ˚O: ⊗ ˚E:]
   - (e) ˚S:O:. ⊗ > [˚S: ⊗ ˚O: ⊗ ˚E:]
   - (f) ˚B:O:. ⊗ > [˚B: ⊗ ˚O: ⊗ ˚E:]
   - (g) ˚T:O:. ⊗ > [˚T: ⊗ ˚O: ⊗ ˚E:]
   - (h) ˚M:O:. ⊗ > [˚M:⊗ ˚O: ⊗ ˚E:]
   - (i) ˚F:O:. ⊗ > [˚F: ⊗ ˚O: ⊗ ˚E:]

Chapter 3

Semantics

«Death and life are in the power of the tongue.» Proverbs 18:20-21

Now we will learn how to produce (and recognize) words, phrases, texts and hypertexts in IEML. The section 3.1 presents the paradigmatic circuits of the dictionary, and the subsequent section 3.2 presents the syntagmatic, textual, and hypertextual circuits and their regular forms of construction.

3.1 The paradigmatic circuits of the dictionary

3.1.1 Some definitions

3.1.1.1 The dictionary

The IEML dictionary includes a basic lexicon and *thesauri* by field. Both the basic lexicon and the thesauri are composed of a set of interoperable paradigmatic networks, called *keys*. The nodes of these networks represent the terms of the dictionary, and the links between these networks represent the semantic relationships between the terms.

3.1.1.2 Difference between rhizome and semantic circuit

The two main differences between a *semantic circuit* and a *rhizome* are that first, a circuit can contain or channel flows of semantic current, and second, a circuit is automatically described as a network of semantic relations between expressions in natural language.

3.1.1.3 Transcoding of a rhizome into a semantic circuit

We can compare the rhizome coded by a script to the stippled outline of a set of possible circuits. Using the information provided by the IEML dictionary, the algorithm for transcoding the Rhizome toward the semantic sphere (see 3.3) transforms certain bulbs of a rhizome into *reservoirs*, transforms certain filaments of this same rhizome into *channels*, and describes the reservoirs and the channels in natural language.

3.1.1.4 Complexity of the semantic circuits

Thus, a script that is equipped with the dictionary can automatically generate a complex semantic circuit that articulates, notably, paradigmatic circuits, syntagmatic circuits, textual circuits and inter-textual circuits.

The purpose of this chapter is to explain the method for constructing the paradigmatic circuits of the dictionary, which make it possible to automatically give meaning to IEML scripts.

3.1.2 Method for constructing original keys

The paradigmatic circuits are constructed from elementary circuits called paradigmatic keys or simply *keys*. First I will explain how to construct *original keys* by following a semi-automatic method, and then how to assemble these keys into coherent keychains. Then we shall see how to create however many *derived keys* (analogous and specialized) from original keys.

3.1.2.1 General structure of the method for construction of original keys

To construct an original key according to the semi-automatic method:

1. we apply a key formatting algorithm to a script that represents a category of layer Ln,
2. we select the filaments that are *not to be* transformed into channels, and the bulbs that are *not to be* transformed into reservoirs,
3. we describe the reservoirs in natural languages, which makes them into *terms*.

3.1.2.2 Algorithm for construction of keys

The algorithm for the construction of keys is similar to the algorithm for the construction of rhizomes (see 2.5), with these two differences:

1. it is limited to the layer Ln[1],
2. it transforms the bulbs into reservoirs and the filaments into channels.

3.1.2.3 Selection of which filaments and which bulbs are not to be opened

By default, all the filaments are transformed into channels and all the bulbs are transformed into reservoirs. To obtain the desired key, we need only select which bulbs and filaments are *not to be* opened.

[1]I.e., to point 1 of 2.5.1.

CHANNELS	**Multiplication** $\otimes$	**Addition** $\oplus$
Order $>$	$\otimes >$ etymological relation	$\oplus >$ taxonomic relation
Symmetry $\sqcap$	$\otimes\sqcap$ relation of complementarity	$\oplus\sqcap$ relation of concurrence

Table 3.1: The four paradigmatic channels between the terms of original keys

Restriction of the channels Concerning the filaments, it is possible to cancel *only* the transformation of the *multiplicative* filaments into channels.

1. For the filaments of multiplicative *symmetry*, we need only cancel the transformation for the whole of the key (see 2.5.2 point 6).
2. For the filaments of multiplicative *order* we choose one, two, or three *roles* (substance, attribute, mode) for which the transformation does not take place, and this applies to the whole of the key.

Restriction of the reservoirs To select which bulbs not to transform into reservoirs, we simply list them. If the bulb selected not to be transformed into a reservoir is located on the route of a relation of additive order between reservoirs, this relation "jumps" or "crosses" the eliminated bulb in the form of a channel of additive order. On the other hand, the non-transformation of a bulb into a reservoir interrupts the multiplicative channels that might have passed through it.

3.1.2.4 Relations between the terms of original keys

When they are described in natural languages, the reservoirs of keys are called *terms,* and the channels then represent the*paradigmatic relations* between the terms. The four types of paradigmatic relations between the terms of original keys are summarized in the table 3.1.

1. The relations of multiplicative order are interpreted as *etymological* relations. Thus, the terms from layer Ln–1 representent the "roots" of the terms of layer Ln. The meaning of the terms from layer Ln derives from the meaning of its lower-layer roots, even if one can never *deduce* the meaning of the terms from that of their roots. For example *s.u.-m.u.-'** (dream) derives from the meaning of *s.u.-** (image) and from the meaning of *m.u.-** (symptom).
2. The relations of multiplicative symmetry indicate semantic *complementarities.* For example, *s.u.-m.u.-'** (dream) and *m.u.-s.u.-'** (fantasy) are complementary.

3. The relations of additive order signal *taxonomic* relations, in other words the relations of sets to sub-sets. For example *s.u.-m.u.-'** (dream) belongs to *M:M:.u.-M:M:.u.-'** (semiotic functions). Note that *M:M:.u.-M:M:.u.-'** contains 81 singular sequences.

4. Finally, the relations of additive symmetry indicate relations of *concurrence*. This idea of concurrence can be understood in two senses. First, in the sense that the choice of a given term to appear in a statement equates to the elimination of the competing terms. Second, in the sense that the concurrent terms together "concur" in the definition of the term that gathers them. For example, in the key *M:M:.O:M:.-**, the term *m.a.-** (parent) has as its concurrents the other terms from the column *m.O:M:.-** (actualized affect) such as *m.y.-** (emotional intelligence) and *m.u.-** (symptom) as well as the terms from the row *M:M:.a.-** (social function) such as *b.a.-** (storyteller) and *f.a.-** (healer).

3.1.2.5 Super-keys

The elementary taxa (or final leaves of the additive tree) of an ordinary key correspond to singular sequences, such as *s.u.-m.u.-'**. By contrast, *super-keys* are keys whose elementary taxa are the most general taxa (or final roots of the additive tree) of ordinary keys, such as *M:M:.u.-M:M:.u.-'**.

For example *(M:M:.a.-M:M:.a.-' + M:M:.u.-M:M:.u.-' + M:M:.i.-M:M:.i.-')** (collective intelligence actualization functions) is a super-key whose three elementary taxa are *M:M:.u.-M:M:.u.-'** (semiotic functions), *M:M:.a.-M:M:.a.-'** (social functions) and *M:M:.i.-M:M:.i.-'** (technical functions). Several levels of super-key may exist , but the keys, their super-keys, their super-super-keys, and so forth are necessarily of the same layer.

3.1.2.6 The keychain of interoperable keys from the basic lexicon

A paradigmatic circuit is interoperable if it contains only one natural-language description of the same term and if the same description is not given to distinct terms. The *basic lexicon* is a keychain of distinct, original keys that together form an interoperable paradigmatic circuit. The keys in the basic lexicon communicate among themselves only by channels of multiplicative order (etymology) and super-keys. The paradigmatic circuit of the basic lexicon is not necessarily connected.

This *basic lexicon* includes all the interoperable keys and super-keys that are transverse to all the universes of discourse and that may be used everywhere.

3.1.3 The key *O:O:.**

If we apply the key formatting algorithm described in 3.1.2.2 to *O:O:.**, we automatically obtain the set of all nine reservoirs : *wo. / wa. / wu. / we. / U:O:. / A:O:. / O:U:. / O:A:. / O:O:.**. We also obtain the circuit of the 48 channels described below. In the case of *O:O:.**, there is no need to select

	O:O:. generate	
	O:U:. generate into the virtual	O:A:. generate into the actual
U:O:. generate from the virtual	U:U:E:. wo. reflect	U:A:E:. wa. operate
A:O:. generate from the actual	A:U:E:. wu. perceive	A:A:E:. we. reconstitute

Figure 3.1: The key *O:O:.** (Generate)

which bulbs or filaments not to open and, as we shall see, all the reservoirs are terms described in natural languages. Figure 3.1 shows a compact representation of the principal relations between the terms, and attention to the figure will make it easier to understand the explanations that follow here below.

3.1.3.1 Paradigmatic channels

Paradigmatic relations are automatically defined as follows by the key formatting algorithm of 3.1.2.2, which itself derives from the Rhizome construction algorithm from the Script, as explained in 2.5.

- Channels of multiplicative order (etymology)
  wo. ⊗ > [U: ⊗ U: ⊗ E:]
  wa. ⊗ > [U: ⊗ A: ⊗ E:]
  wu. ⊗ > [A: ⊗ U: ⊗ E:]
  we. ⊗ > [A: ⊗ A: ⊗ E:]
  U:O:. ⊗ > [U: ⊗ O: ⊗ E:]
  A:O:. ⊗ > [A: ⊗ O: ⊗ E:]
  O:U:. ⊗ > [O: ⊗ U: ⊗ E:]
  O:A:. ⊗ > [O: ⊗ A: ⊗ E:]
  O:O:. ⊗ > [O: ⊗ O: ⊗ E:]

Remember that each of the nine preceding lines represents three channels of multiplicative order, which makes a total of 27 channels of multiplicative order (or etymological relations).

- Channels of multiplicative symmetry (complementarity)
  wu. $\otimes\sqcap$ wa.
  U:O:. $\otimes\sqcap$ O:U:.
  A:O:. $\otimes\sqcap$ O:A:.

Thus, the key O:O:. includes three channels of multiplicative symmetry. In the figure 3.1, the two cases *wo.** and *we.** mark the diagonal "axis of symmetry" of the relation of multiplicative symmetry wu. $\otimes\sqcap$ wa.

- Channels of additive order (taxonomy)
  U:O:. $\oplus >$ wo.
  U:O:. $\oplus >$ wa
  A:O:. $\oplus >$ wu.
  A:O:. $\oplus >$ we.
  O:U:. $\oplus >$ wo.
  O:U:. $\oplus >$ wu.
  O:A:. $\oplus >$ wa.
  O:A:. $\oplus >$ we.
  O:O:. $\oplus >$ U:O:.
  O:O:. $\oplus >$ A:O:.
  O:O:. $\oplus >$ O:U:.
  O:O:. $\oplus >$ O:A:.

Thus, the key includes 12 channels of additive order.

- Channels of additive symmetry (concurrence)
  [wo. $\oplus\sqcap$ wa.]
  [wu. $\oplus\sqcap$ we.]
  [wo. $\oplus\sqcap$ wu.]
  [wa. $\oplus\sqcap$ we.]
  [U:O:. $\oplus\sqcap$ A:O:.]
  [O:U:. $\oplus\sqcap$ O:A:.]

Thus, the key includes six channels of additive symmetry.

*O:O:.** constitutes the simplest of all the original keys, and we see that it already includes nine terms and 48 paradigmatic channels, of which 27 (the etymological relations) concern the relations between the terms of the key and the terms of a key from a lower layer (here, layer 0). For all the examples that follow, I will not explicitly list all the channels, because I will suppose that the reader understands their automatic mode of construction. Rather, I will give one or several matrix-like tables of the 3.1 type, and I will point out those bulbs not transformed into reservoirs and those filaments not transformed into channels. Having done so, I will concentrate my exposé, as synthetically as possible, on the *semantic relations between the terms*, which relations are modelled on the paradigmatic channels.

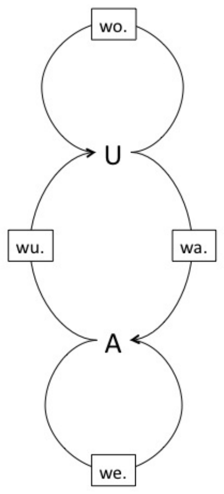


Figure 3.2: The 4 singular terms of *O:O:.**

3.1.3.2 Semantic relations between the terms

I call *singular term* a term that corresponds to a sole sequence from the six elementary symbols. The heart of the O:O:. key is found in the relation between its four singular terms, a relation that traces a generative circle (see figure 3.2).

1. *wa.** (operate) corresponds to a passage or to a transformation from the virtual to the actual. Here we suppose that the operation involved starts from a virtual or mental plan or vision and that it is accomplished or projected into an actual, physical domain.

2. *wu.** (perceive) corresponds to an inverse transformation to that of *wa.** in which an actual reality is transferred to a virtual plan, in the mind, in the form of an image of the actual.

3. *wo.** (reflect) corresponds to a transformation or to a renewal within the mind. *wo.** regenerates the virtualities, such that the whole of the cycle is in effect productive or creative.

4. *we.** (reconstitute or react) corresponds to a causal transformation that is internal to the physical environment and that, in reconstituting itself after the action, responds in a manner that may be unpredictable, surprising, or undesired to the operation of *wa.**.

We can understand *O:O:.** (generate) as the fundamental structure of the cycle sensorial *wu.** / motor *wa.**, provided we allow in this cycle transformations that are internal to the thinking subject *wo** and to the physical

	O:M:. act		
	O:S:. act according to sign	O:B:. act according to being	O:T:. act according to thing
U:M:. act virtually	U:S:E:. y. know	U:B:E:. o. want	U:T:E:. e. can
A:M:. act actually	A:S:E:. u. state	A:B:E:. a. undertake	A:T:E:. i. do

Figure 3.3: The key *O:M:.**

environment *we.**. In IEML we can use verbs that signify not only one of the four phases of the cycle, but also:

1. the four phases together: *O:O:.** (generate, running the entire cycle),
2. the two phases that begin in the virtual: *U:O:.** (generate from the virtual, in other words «reflect and act»),
3. the two phases that begin in the actual: *A:O:.** (generate from the actual, in other words “reconstitute and perceive”),
4. the two phases that terminate in the virtual: *O:U:.** (generate toward the virtual, in other words “perceive and reflect”)
5. the two phases that terminate in the actual: *O:A:.** (generate toward the actual, in other words «operate and reconstitute”, a verb that includes an action and its consequences).

The reader can verify that the 48 paradigmatic channels are indeed based on the semantic relations between terms.

3.1.4 The key *O:M:.**

As in the preceding case, the complete paradigmatic circuit is obtained automatically and without restriction by the algorithm described in 3.1.2.2 from the entry *O:M:.**. Because its substance is *O:**, like the preceding key, *O:M:.** is a verb and contains only verbs. Yet this time, the attribute is *M:**, and this is why the object or the domain of the action will be more precise.

3.1.4.1 The six singular terms

The six singular terms of the key are organized as follows:

When the substance is *U:**, as in *U:M:.**, there is a virtuality or a modality to the action[2]. Three modalities can be distinguished:

y.* (to know) corresponds to the attribute "sign" because it includes representations;

o.* (to want) corresponds to the attribute "being" because it implies the intentional core of the person;

e.* (to be able) corresponds to the attribute "thing", because it refers to a capacity to transform reality.

*U:M:.** (to act virtually) includes the three modalities and thus signifies: to know, to want, and to be able.

When the substance is *A:**, as in *A:M:.** then an effective action is involved. Here again, we can distinguish three types of effective acts:

u.* (to state), whose attribute is "sign", evokes expression, speech, enunciation;

a.* (to commit), whose attribute is "being" signifies commitment, personal involvement;

i.* (to do), whose attribute is "thing", denotes physical action, the material act.

*A:M:.** (to act actually) includes the three types of effectuation and thus signifies: state, commit, and do.

*O:M:.** (to act generally) therefore signifies: know, state, want, commit, be able and do. All the dimensions of the act are implied here.

3.1.4.2 The two dialectics

In general, the *additive symmetries* of the paradigmatic keys (in other words, the *relations of concurrence* described in 3.1.2.4 under point 4) are organized by leaving invariable a substance seme and varying the attribute seme or by setting an attribute seme and varying the substance seme. This procedure is shown in figure3.4, where the beginning of the arrows marks the substance seme, and their end the attribute seme.

[2]See, from Algirdas Julien Greimas and Joseph Courtès, *Sémiotique : dictionnaire raisonné de la théorie du langage*, Hachette, 1979 and from Algirdas Julien Greimas, *Du sens, 2*, Seuil, Paris, 1983.

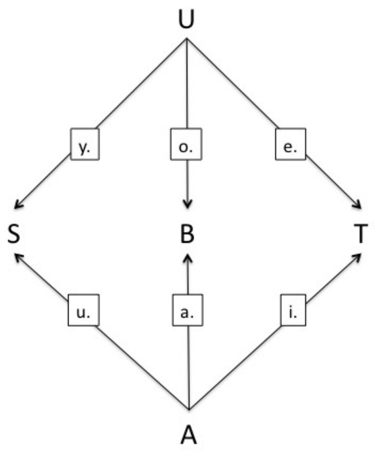


Figure 3.4: The six singular terms of *O:M:.**

- **Dialectic sign / being /** thing
  If the substance seme is set as *U:** and the attribute seme is varied to *S: / B: / T: **, then with "know / want / be able" we obtain a dialectic of sign / being / thing. Likewise, if we set the substance seme as *A:** and the attribute seme as *S: / B: / T: **, then again using "state / commit / do" we obtain a dialectic of sign / being / thing.

- **Dialectic virtual / actual**
  By contrast, if we set the attribute seme as *S:** and vary the substance seme to *U: / A: **, then with *O:S:.** "know / state" we obtain a virtual / actual dialectic. In the same way, we obtain such a virtual / actual dialectic with *O:B:.** "want / commit" and *O:T:.** "be able / do".

3.1.5 The key *E:F:.O:M:.**

Our third and last example is provided by the key *E:F:.O:M:.-**. Contrary to the two preceding keys, this is not a verbal paradigm but an auxiliary paradigm, because it begins with *E:**. Likewise by contrast to the two examples above, which were at layer L1, *E:F:.O:M:.-** is found at layer L2. Moreover, the circuit of its channels and reservoirs is generated by the algorithm described in 3.1.2.2 starting from the entry *E:F:.O:M:.-**, but with the restriction [3] that there are no channels of multiplicative order (and thus no etymological relation

[3] See 3.1.2.3 point 2.

		E:F:.O:M:.- conjugation				
		E:O:.O:M:.- participation mood		E:M:.O:M:.- tense /time / when		
		E:U:.O:M:.- passive	E:A:.O:M:.- active	E:S:.O:M:.- future	E:B:.O:M:.- present	E:T:.O:M:.- past
E:F:.U:M:.- delarative mood	E:F:.y.- indicative mood	E:U:.y.- known	E:A:.y.- knowing	E:S:.y.- will know	E:B:.y.- know now	E:T:.y.- has known
	E:F:.o.- optative mood	E:U:.o.- wanted	E:A:.o.- willing	E:S:.o.- will want	E:B:.o.- want now	E:T:.o.- would
	E:F:.e.- empowering mood	E:U:.e.- enabled	E:A:.e.- enabling	E:S:.e.- will be able	E:B:.e.- can now	E:T:.e.- could
E:F:A:M:.- performative mood	E:F:.u.- expressive mood	E:U:.u.- expressed	E:A:.u.- expressing	E:S:.u.- will express	E:B:.u.- express now	E:T:.u.- has expressed
	E:F:.a.- promissive mood	E:U:.a.- commited	E:A:.a.- committing	E:S:.a.- will commit	E:B:.a.- commit now	E:T:.a.- has committed
	E:F:.a.- imperative mood	E:U:.i.- be done	E:A:.i.- be doing	E:S:.i.- will execute	E:B:.i.- execute now	E:T:.i.- should

Figure 3.5: The key *E:F:.O:M:.-**

with the terms of the immediately lower layer) that correspond to *substance* semes.

3.1.5.1 Invariance of the seme in substance role and variation of the seme in attribute role

First we will consider the eight boxes on the left side of figure 3.5, in which all the terms invariably have *E:F:.** as substance. Here we will concentrate on the variations in meaning, which concern the semes in attribute role. First of all, we see that the attribute of *E:F:.O:M:.-** is *O:M:.**, which was just analyzed in 3.1.4. I used *y. / o. / e. / u. / a. / i.** to indicate the *mode* of the verbs.

Because *y.** signifies "know", we will use it to signify the indicative. For example, "to know that one can" will be written *e.-E:.-E:F:.y.-'**[4]. Using the indicative in IEML consists simply of saying that one knows what one is saying.

[4]In this example, as in those that follow, the root of the verb will be in the role of substance, the conjugation in the role of mode, and the role of attribute will be empty. See 3.2.4.2 for

Because *o.** signifies “want”, we use it to signal the optative, as in “to want to reflect” or “to choose to reflect” : *wo.-E:.-E:F:.o.-’**.

Given that *e.** signifies “to be able”, we will use it to denote capacity or enablement. For example “to be able to express” would be said thus: *u.-E:.-E:F:.e.-’**.

The indicative, optative and habilitative modes are organized into a super-mode *E:F:.U:M:.-** which I have named declarative, which indicates simply that it concerns the three modalities (know, want, be able) of action.

The declarative super-mode is in a concurrence relation[5] with another super mode, called performative, which includes 1) the expressive or enunciative mode based on m*u.** (express or state), 2) the promissive mode based on *a.** (to commit) and 3) the imperative mode based on *i.** (do). In the case of this last mode, it is as if the verb in the imperative was accompanied by the mention “to be done”. The imperative implies of course the notion of obligation.

Even if they are not visible in figure 3.5, other groupings into super-modes are authorized by the key generating algorithm. For example, we can use the super-mode *E:F:.O:T:.-**, which signifies “be able *and* must”, as in *wa.-E:.-E:F:.O:T:.-’ ** (be able and must operate), and so on.

3.1.5.2 Invariance of the seme in attribute role and variation of the seme in substance role

We will now examine the eight boxes on the top of figure 3.5. Here, all the terms invariably have *O:M:.** in the attribute role, and the significant variation thus concerns *E:F:.** in the role of substance. However, as *E:** cannot be declined because it contains only a single primitive, the variation can in reality relate only to the attribute of the substance, namely *F:**. And we know that *F:** automatically decomposes into *O:** and *M:**.

Because *O:** represents a two-poled dialectic *U: / A:** and we are now in the context of conjugation, I have chosen to use it to represent the dialectic between a passive mode *U:** and an active mode *A:**. The attribution of activity to the actual *A:** is self-explanatory, leaving only the virtual *U:** to represent the passive mode. We see that the notion of “mode” here extends in a direction perpendicular to the one used to characterize the indicative, the optative, the habilitative, the expressive, the promissive and the imperative. In effect, each of these latter modes can be declined in the active and in the passive. For example, one can be enabled, capable (passivity), *or* enabling, rendering capable (activity). Therefore, I have named *E:O:.O:M:.-** “the mode of participation”, it being understood that participation can be active or passive.

Given that *M:** represents a dialectic at three poles, it was almost inevitable, in the context of conjugation, to use it to indicate the three main verb tenses: the future *S:**, the present *B:** and the past *T:**. I first attributed the past to the “thing” because the past is necessarily reified, and the present to “the being”. All that remained therefore for the future was the “sign”.

---

the manner of constructing words.

[5]See 3.1.2.4, point 4.

Thus, *E:M:.O:M:.-** represents “the time”, in other words the dialectic future / present / past.

3.1.5.3 Matrix of the singular terms

The thirty inside boxes of the figure 3.5 contain all the singular terms and the meaning of each term is taken automatically from the column (active / passive / future / present / past) and from the row (indicative / optative/ habilitative / expressive / promissive / imperative) at the intersection of which it is located. Remember that it is always possible to add several terms in order to indicate exactly the nuance that one has in mind. For example, *wa.- E:.- (E:B:.a.- + E:T:.o.-)’** signifies “to have wanted and to now commit to operate”.

3.1.5.4 To conclude this list of examples: heuristic procedures

In the following chapters I will give examples of keys drawn from the IEML dictionary. Yet even before that, the three examples just seen have already shown how the algorithm for the automatic construction of keys lends itself to the semantic arrangement of terms and their paradigmatic relations. They also enabled me to present the main heuristic procedures for the design of keys, including notably:

1. the creative use of the dialectics E/F, O/M, U/A and S/B/T;
2. the semantic play on the variations in a syntactic role when another role remains invariant;
3. the reuse of the terms from layer Ln–1 as semes of the terms from layer Ln.

Figure 3.6 offers a summary of essential information on original keys.

3.1.6 Method of construction for derived keys

3.1.6.1 Thesauri derived from the basic lexicon

In order to itemize the important concepts from a particular theoretical field, practical domain or universe of discourse, there are two possible methods. Either create original keys (belonging to the basic lexicon), or create keys that are *distinct* from the original keys but *derived* from them and semantically interoperable with them. The derived keys include two types of keys: analogous keys and special keys. In this way, the dictionary can be augmented as desired not only by new original keys but also by keys derived from original keys.

3.1.6.2 Analogous keys

Generalities Analogous keys are identical to those from the basic lexicon, *with the sole exception of their descriptors.* This means that the form of the circuit (channels and reservoirs) remains exactly the same as in the original key

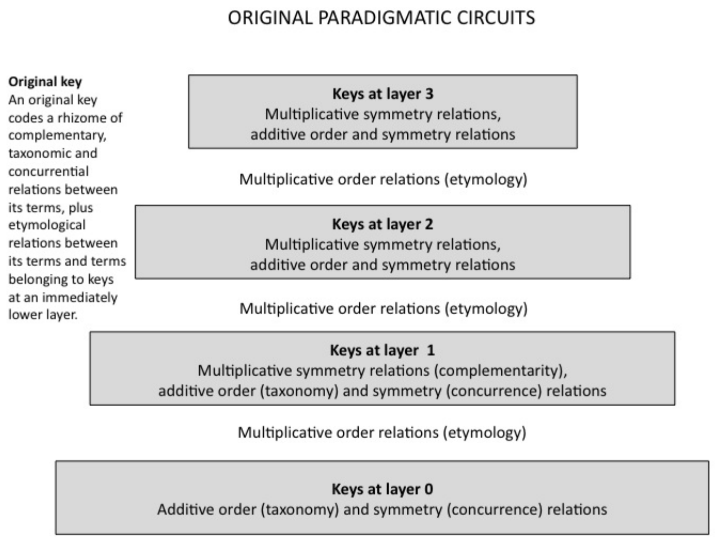

ORIGINAL PARADIGMATIC CIRCUITS
Original key
An original key codes a rhizome of complementary, taxonomic and concurrential relations between its terms, plus etymological relations between its terms and terms belonging to keys at an immediately lower layer.
Keys at layer 3
Multiplicative symmetry relations, additive order and symmetry relations
Multiplicative order relations (etymology)
Keys at layer 2
Multiplicative symmetry relations, additive order and symmetry relations
Multiplicative order relations (etymology)
Keys at layer 1
Multiplicative symmetry relations (complementarity), additive order (taxonomy) and symmetry (concurrence) relations
Multiplicative order relations (etymology)
Keys at layer 0
Additive order (taxonomy) and symmetry (concurrence) relations

Figure 3.6: Original keys

and the only thing that changes is the set of the natural-language descriptors applied to the terms. The same key from the basic lexicon can be modified according to the semantic needs of several *thesauri*. But the different analogous keys from the same original key share an identical structure of semantic relations, which indeed is that of the original key. An analogous key of degree 1 can in turn serve as the foundation for the construction of an analogous key of degree 2, and so on recursively. In order to maintain interoperability, which requires that the same IEML terms not have several natural-language descriptors, we differentiate derived-key terms from original-key terms by means of a *thesaurus address*. To avoid all ambiguity, this address must figure in the USLs that contain the terms of an analogous key (see below, under section 3.1.7).

Example of analogous key We can imagine for example a "thesaurus of time" in which *O:O:.* would represent the four main phases of a luminous cycle : *wo.** would represent "shine", *wa.** "disappear, go out", *we.** "be extinguished", and *wu.** "appear, light up". In a second degree of analogy, these verbs can be applied to the phases of the moon (full, waning, new, waxing), to the circadian rhythm (day, dusk, night, dawn), etc. Thus, the structure of *O:O:.** could be used for numerous *cycles of action in four phases.* Examples of analogous keys will be available in the on-line dictionary once published.

Order and analogous symmetry There exists a relation of analogous order [≀ >], or relation of comparison between a root key of degree n–1 and a leaf key

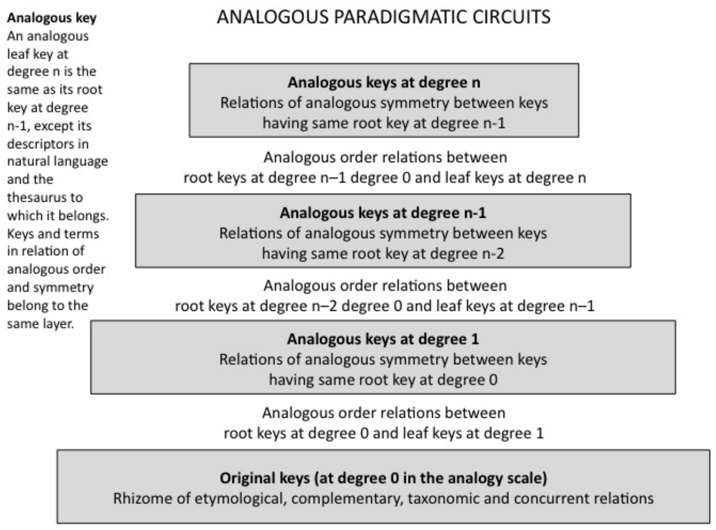

Analogous key
An analogous leaf key at degree n is the same as its root key at degree n-1, except its descriptors in natural language and the thesaurus to which it belongs. Keys and terms in relation of analogous order and symmetry belong to the same layer.
ANALOGOUS PARADIGMATIC CIRCUITS
Analogous keys at degree n
Relations of analogous symmetry between keys having same root key at degree n-1
Analogous order relations between root keys at degree n–1 degree 0 and leaf keys at degree n
Analogous keys at degree n-1
Relations of analogous symmetry between keys having same root key at degree n-2
Analogous order relations between root keys at degree n–2 degree 0 and leaf keys at degree n–1
Analogous keys at degree 1
Relations of analogous symmetry between keys having same root key at degree 0
Analogous order relations between root keys at degree 0 and leaf keys at degree 1
Original keys (at degree 0 in the analogy scale)
Rhizome of etymological, complementary, taxonomic and concurrent relations

Figure 3.7: Relations of order and of analogous symmetry

of degree n. There exists the same relation of analogous order between the corresponding terms from the root keys and the leaf keys.

There exists a relation of analogous symmetry [≀⊓], or relation of resemblance between the leaves of degree n having the same root of degree n–1. There exists the same relation of analogous symmetry between the corresponding terms from the leaf keys of degree n having the same root key of degree n–1.

Figure 3.7 summarizes the essential information on relations of order and analogous symmetry.

3.1.6.3 Special keys

Generalities Special keys are constructed from lexicalized statements. Lexicalized statements can be morphemes (additions of terms from the basic lexicon), words (multiplications of morphemes), phrases (multiplications of words) and so on. One (or more) of the grammatical roles of the statement can be played alternatively by an original (or analogous) term chosen from a list whose members are in a relation of opposition or of possible substitution. The special terms are qualified by a descriptor in natural language, proper to a particular thesaurus.

Example of a special key Let us suppose, for example, that we want to represent the «three states of the Me» (Child, Parent, Adult) from *transactional analysis*[6]. We shall use the following terms, taken from original keys in the basic

[6]A psychotherapy school.

lexicon.

Terms

E:E:U:.	attribute
E:E:A:.	genitive
E:S:.wo.-	me
E:T:.c.-	state
E:F:.wa.-	plural
m.a.-	parent
d.a.-m.a.-f.o.-'	child
n.a.-s.a.-f.o.-'	adult

The structure of the specialized key is as follows :

*E:T:.c.-' [E:F:.wa.-' | m.a.-' | d.a.-m.a.-f.o.-' | n.a.-s.a.-f.o.-'] E:E:U:.-', E:S:.wo.-', E:E:A:.-',_**

The four specialized terms of this key are as follows :

*E:T:.c.-' E:F:.wa.-' E:E:U:.-', E:S:.wo.-', E:E:A:.-',_** **Three states ego**[7]
*E:T:.c.-' m.a.-' E:E:U:.-', E:S:.wo.-', E:E:A:.-',_** **Parent ego state**
*E:T:.c.-' d.a.-m.a.-f.o.-' E:E:U:.-', E:S:.wo.-', E:E:A:.-',_** **Child ego state**
*E:T:.c.-' n.a.-s.a.-f.o.-' E:E:U:.-' E:S:.wo.-' E:E:A:.-',** **Adult ego state**

Relations of order and of specialized symmetry Relations of specialized order [$\mp >$], or relations of construction connect the terms of degree n–1 and the terms of degree n that result from their multiplication. For example, *E:T:.c.-' m.a.-' E:E:U:.-', E:S:.wo.-', E:E:A:.-',_** (Ego State Parent) is superior to *m.a.-** (parent) according to the specialized relation of order. The relations of specialized order always connect terms from distinct layers. A key or a specialized term can recursively be used in a specialized key of superior order.

The relations of specialized symmetry [$\mp\sqcap$], or relations of opposition connect terms that occupy the same grammatical role in the same specialized key.

Figure 3.8 summarizes the essential information about relations of order and of specialized symmetry.

[7]The grammatical structure of the proposition is the following: the state *E:T:.c.-** has the attribute *E:E:U:.** "parent + child + adult" *(m.a.-' + n.a.-s.a.-f.o.-' + d.a.-m.a.-f.o.-')**, and this state is that of *E:E:A:.** me *E:S:.wo.-**. In chapter 3.2 below, we will study in greater detail the gramatical construction of propositions.

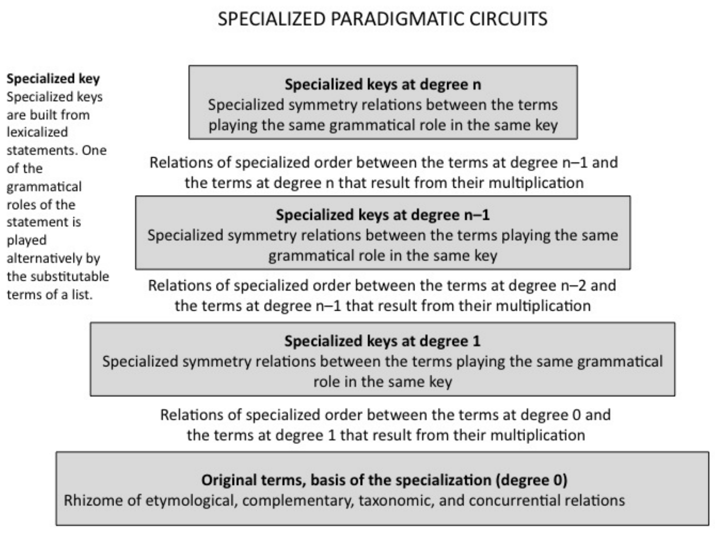

SPECIALIZED PARADIGMATIC CIRCUITS
Specialized key
Specialized keys are built from lexicalized statements. One of the grammatical roles of the statement is played alternatively by the substitutable terms of a list.
Specialized keys at degree n
Specialized symmetry relations between the terms playing the same grammatical role in the same key
Relations of specialized order between the terms at degree n–1 and the terms at degree n that result from their multiplication
Specialized keys at degree n–1
Specialized symmetry relations between the terms playing the same grammatical role in the same key
Relations of specialized order between the terms at degree n–2 and the terms at degree n–1 that result from their multiplication
Specialized keys at degree 1
Specialized symmetry relations between the terms playing the same grammatical role in the same key
Relations of specialized order between the terms at degree 0 and the terms at degree 1 that result from their multiplication
Original terms, basis of the specialization (degree 0)
Rhizome of etymological, complementary, taxonomic, and concurrential relations

Figure 3.8: Relations of order and of specialized symmetry

CHANNELS	**Multiplication** $\otimes$	**Addition** $\oplus$
Order $>$	$\otimes >$ etymological relation	$\oplus >$ taxonomic relation
Symmetry $\sqcap$	$\otimes\sqcap$ relation of complementar-ity	$\oplus\sqcap$ relation of concurrence

Table 3.2: The 4 types of original paradigmatic relations

3.1.7 The dictionary

3.1.7.1 Recapitulation of paradigmatic circuits

In summary, there exist two major types of keys : original keys and derived keys. Among the original keys, we can distinguish beyween ordinary keys and super-keys. Among the keys derived from original keys, we can distinguish analogous keys and special keys. The *dictionary* includes original keys as well as derived keys (analogous or special), and the basic lexicon as well as the thesauri.

CHANNELS	**Analogy** $\wr$	**Specialization** $\mp$
Order $>$	$\wr >$ relation of comparison	$\mp >$ relation of construction
Symmetry $\sqcap$	$\wr\sqcap$ relation of resemblence	$\mp\sqcap$ relation of opposition

Table 3.3: The 4 types of derived paradigmatic relations (analogous and specialized)

The terms of the dictionary are formalized by reservoirs, and the relations by channels . These terms maintain among them relations of order and of additive and multiplicative symmetry, both analogous and specialized. Thus, in all there exist eight types of paradigmatic channels, as summarized in tables 3.2 and 3.3.

3.1.7.2 Method of construction of a thesaurus address

Each thesaurus is addressed by a *unique IEML proposition that draws its terms from the basic lexicon.* The proposition that indicates the address begins with *o.h.n.-** ("choose an interpretation" *o.h.-** bearing the flexion "world" *n.**) and ends with *E:E:T:.** which marks the relation of object complement between the verb of substance *o.h.n.-** and the unit of meaning in attribute as indicated in the particular thesaurus.

If a USL can be interpreted according to several *thesauri* then several thesaurus addresses will be given in that USL. The terms of that USL that do not appear in the *thesauri* indicated by the USL are obviously interpreted according to the original keys.

Example from the thesaurus of social criticism Let us suppose for example that we want to indicate that a USL must be interpreted according to the thesaurus of «social criticism» (*m.-o.-s.y.-'** in the basic lexicon). We shall then have the category *o.h.n.-'m.-o.-s.y.-'E:E:T:.-',** appear in that USL.

Example from the thesaurus of time In order to represent the address from the "thesaurus of time" discussed in 3.1.6.2, we could write *o.h.n.-'t.o.-n.o.-'E:E:T:.-',**, *t.o.-n.o.-'** signifying regular calendar time.

Example from the thesaurus of transactional analysis To return to the example from transactional analysis seen in the preceding sub-section, the thesaurus address here:

*o.h.n.-', s.a.-f.a.-' j.p.- a.a.- E:E:A:.-' E:E:A:.-', E:E:T:.-',_** combines four terms:

Terms

j.p.-	new analysis
a.a.-	contract, strike an agreement
s.a.-f.a.-'	psychotherapy
E:E:A:.	genitive

3.2 Enunciative circuits

3.2.1 Units of meaning

3.2.1.1 The notion of units of meaning

In the paradigmatic circuits as well as in syntagmatic, textual or inter-textual circuits, we no longer are dealing with categories and the corresponding rhizomes (as in the Script before any interpretation in natural languages) but rather with *units of meaning*, in other words with reservoirs that are interconnected by channels and described in natural languages according to keys (whether these keys be original, transposed, or specialized).

3.2.1.2 Units in language and units in speech

Two main types of units of meaning can be distinguished:

1. units "in language", which are the terms from the dictionary interconnected by *paradigmatic* circuits;

2. units "in speech" which are the propositions, texts and hypertexts that are connected, respectively, by the syntagmatic, inter-propositional and intertextual circuits. The units of meaning "in speech" correspond to *enunciative circuits*, in other words to the syntagmatic (propositions), inter-propositional (texts or USL) and intertextual (hypertexts) circuits.

3.2.1.3 Terms

As we have seen in chapter 3.1, *terms* are the elementary units of meaning that serve at the foundation for the construction of all other units of meaning. The dictionary connects the terms among themselves by means of paradigmatic channels that describe their semantic relations "in language" (see 3.1.2.4 and 3.1.7.1 for the semantics of these channels).

Propositional units	Construction by isolayer operations
Morpheme	(term $i \oplus$ term j...)
Word	(morpheme $i \otimes$ morpheme j)
Clause of degree D1	(word $i \otimes$ word $j \otimes$ word k)
Phrase of degree D1	(clause i D1 $\oplus$ clause j D1...)
Clause of degree Dn	(ph. i Dn–1 $\otimes$ ph. j Dn–1 $\otimes$ ph. k Dn–1)
Phrase of degree Dn	(clause i Dn $\oplus$ clause j Dn...)

Table 3.4: Hiearchy of propositional units

3.2.1.4 Propositions

General description *Propositions* describe the semantic relations between terms *in* statements, in other words "in speech". In propositions, terms are added to produce *morphemes*, morphemes are multiplied to produce *words*, words are multiplied to produce *clauses*, clauses are added to produce*phrases*, and in this way we can produce *phrases of superior degrees*, until the limit of layer L6 is reached.

The recursive operative relations (additive and multiplicative) just described between terms command syntagmatic circuits. The reservoirs of these syntagmatic circuits are, respectively, morphemes, words, clauses and phrases of successive degrees, while the channels between the reservoirs describe their grammatical relations.

Detail of the hierarchy of the propositional units The table 3.4 here below synthetically presents the hierarchy of propositional units.

- A morpheme is constructed by an addition of terms.
- An inflected word is constructed by a multiplication between a root morpheme and a flexion morpheme.
- Clauses of degree D1, which are multiplications of words, describe the grammatical relation between a word in substance role and a word in attribute role by means of a word in mode role.
- Phrases of degree D1, which are additions of clauses, describe a grammatical network of words.
- Clauses of degree D2, which are multiplications of phrases, describe the grammatical relation between a phrase in substance role and a phrase in attribute role by means of a phrase in mode role and so on until the phrases of the last degree Dn.

A morpheme can only consist of a single term, a word of a single morpheme, a clause of a single word, a phrase of a single clause and so on. Thus, a proposition obviously can only consist of a single word.

Rule of empty series When constructing propositions, *one can always add or multiply units of meaning from different layers*, provided one adds *empty series* to the units of meaning from lower layers, so that they will thus be promoted to the layers to which belong the highest units of meaning. *The meaning of a unit is not modified by the addition of an empty series.*

For example, if one wishes to say "officer *l.a.-k.a.-f.o.-'** *and* judge *n.a.-**" one writes :

*(n.a.-' + l.a.-k.a.-f.o.-')** with the addition of an empty series to *n.a.-** in order to promote it to layer L3[8].

3.2.1.5 Differences between propositional units and superior units

The main difference between propositional units of meaning, on the one hand, and textual and hypertextual units of meaning, on the other hand, is that the propositional units (morphemes, words, phrases) can only be added or multiplied with units from the same layer. By contrast, there are no constraints *of layer* upon additions between propositions from a same text, nor on the recursive additions and multiplications of texts that construct hypertexts.

3.2.1.6 Texts (USL)

Texts are units of meaning that result from additive relations between propositions. The text can be considered as the result of addition between the propositions that it contains (each proposition is in a relation of additive order with the text that contains it) and the different propositions of a same text are, by default, in mutual relations of additive symmetry.

3.2.1.7 Hypertexts

Finally, *hypertexts* are units of meaning that result from recursive operations of addition and of multiplication between texts. Remember that semantic multiplication is an operation of three variables (substance, attribute, and mode). At the hypertextual level, the role of substance is played by a set of texts in reference (or cited), the role of attribute is played by a set of texts that refer to the texts in substance role (or citing) and the role of mode is played by a set of texts that explain the relation between the texts in reference and the texts that refer thereto (or citation note). Thus, multiplication here models the act of inter-textual reference (or of citation).

Texts (USL) can be assimilated to units of meaning of hypertextual degree 0 and one can therefore construct "hypertextual propositions" of degree 1, 2, 3, 4, 5 and 6. These propositions can be added in order to produce higher-order hypertextual texts, and these hypertextual texts can in turn serve for new constructions of circuits, and so on.

[8]For all the examples in this work, I strongly suggest consulting the PDF form of the dictionary, in order to situate the terms in the keys of semantic relations from which they take their meaning.

Textual and hpertextual units	Construction by free operations from a usl of order 0
Texte (USL)	(proposition $i \oplus$ proposition j...)
Hypertextual clause of degree D1	(USL $i \oplus$ USL j ...) $\otimes$ (USL $k \oplus$ USL l ...) $\otimes$ (USL $m \oplus$ USL n ...)
Hypertext of degree D1	(hyp. clause i D1 $\oplus$ hyp. clause j D1...)
Hypertextual clause of degree Dn	(hyp. i Dn–1 $\oplus$ hyp. j Dn–1...) $\otimes$ (hyp. k Dn–1 $\oplus$ hyp. l Dn–1...) $\otimes$ (hyp. m Dn–1 $\oplus$ hyp. n Dn–1...)
Hypertext of degree Dn	(hyp. clause i Dn $\oplus$ hyp. clause j Dn...)

Table 3.5: Hierarchy of textual and hypertextual units

In summary, as is indicated in table 3.5, hypertexts (recursive circuits of texts) result from the addition and multiplication of USLs in the same manner as phrases are constructed recursively from words.

A multiplication of USLs leading to a hypertextual clause of degree 1 will be of the form :

*USL**:*USL**:*USL**:.

A multiplication of USLs leading to a hypertextual clause of degree 2 will be of the form :

*USL**:*USL**:*USL**:. *USL**:*USL**:*USL**:. *USL**:*USL**:*USL**:.-

In this way, one can climb the hypertextual multiplicative degrees up to the sixth, by using the IEML punctuation marks (: . - ' , _ ;). A USL of USL - or USL of order 1 - is obtained by adding the hypertexts form degrees 1 to 6. In principle, there is nothing to prevent the construction of USLs of order 2, 3, etc. up to n (n being a natural finite integer). Table 3.5 is limited to a description of the hypertextual hierarchy for USLs of order 0.

3.2.2 The three classes

All natural languages contain different *classes* of units of meaning such as : nouns, verbs, adverbs, adjectives, prepositions, conjunctions, markers of gender, number, case, tense, mode, person, etc. The classes of verbal and nominal units are universal. Moreover it must be noted that there exist not only words but also propositions that are verbal or nominal. In IEML, there exist only three classes of units in the paradigmatic and syntagmatic circuits : verbal, nominal and auxiliary units, but these classes can translate all those of natural languages.

3.2.2.1 Verbal class

Units of meaning that begin with *O:** (*U:** or *A:**) are *verbal.* They may be word-verbs or verbal phrases. Verbal units indicate processes, habitus or events.

3.2.2.2 Nominal class

Nominal units begin with *M:** (*S:**, *B:** or *T:**). They indicate entities or qualities. They include nouns or adjectives, as well as nominal propositions.

3.2.2.3 Auxiliary class

Finally, units that begin with *E:** are *auxili*ary units. They include prepositions, pronouns, adverbs, conjunctions, markers of conjugation, of mode, of case, of gender, of number, of person, etc. Placed in the role of mode, auxiliaries modify morphemes to form inflected words, or indicate the relation between the substance unit and the attribute unit in a phrase. *Auxiliary phrases* in the role of mode represent the relation between phrases in substance role and phrases in attribute role.

3.2.2.4 Interpretation of classes according to their syntactic role

Verbal or nominal terms can be used in the role of mode to indicate the "semantic case" of a word.

Here below we find an example for the "declination" of a noun in substance role. *n.o.-n.o.-'** signifies : “human being”.[9]

n.o.-n.o.-'E:.-'wu.wo.-',* “human child” (from the verb *wu.wo.-** “to be a child”)

n.o.-n.o.-'E:.-'wu.wa.-',* “human youth” (from the verb *wu.wa.-** “to be young”)

n.o.-n.o.-'E:.-'wu.wu.-',* “human adult” (from the verb *wu.wu.-** “to be mature”)

n.o.-n.o.-'E:.-'wu.we.-',* “elderly human” (from the verb *wu.we.-** “to be old”)

Likewise, the verbal or nominal units used in the role of mode in a phrase can indicate the type of relation between the substance unit and the attribute unit.

When they are placed in substance or in attribute, auxiliary units of meaning can be taken in a nominal or a verbal sense, depending on the grammatical context indicated by the mode. We have already encountered this situation above. In :

*E:T:.c.-' m.a.-' E:E:U:.-', E:S:.wo.-', E:E:A:.-',_** “Etat du Moi Parent”

the auxiliary *E:T:.c.-** “state” used in the role of substance, is taken in a nominal sense because the grammatical construction implies that *m.a.-'** “parent” qualifies *E:T:.c.-**. Note that *E:E:U:.** marks the attribute of the noun.

[9] In this case, the role of attribute is occupied by an empty sequence; on this subject, see below, under section 3.2.4.2.

3.2.2.5 Rule of decision for attributing a syntagmatic unit to one of the three classes

The first character, or *initial*, of a chain or characters can be *mix*ed, if its primitives do not belong exclusively to one of the three classes. To resolve this problem, the *decision rule* that enables a syntagmatic unit of meaning to be attributed to one of the three classes is as follows :

`E+O` $\Longrightarrow$ `O`
`E+M` $\Longrightarrow$ `M`
`O+M` $\Longrightarrow$ `M`

For example, *(E:+U:+B:)** will be considered a nominal unit.

3.2.3 The three roles

3.2.3.1 Introduction to the three roles

Contrary to the three classes of units of meaning (verb, noun and auxiliary), the three syntactic roles in IEML (substance, attribute and mode) do not indicate the *nature* of the units of meaning but rather their grammatical *function*, and we have just seen that function has priority over nature. As a general rule, a unit, whatever its class, can play any of the three roles in a higher-layer unit.

What is the sense of these «places» that repeat themselves, fit fractally one inside the other and refer to the three variables of semantic multiplication ? Our three roles describe fundamentally *the three factors of an act of predication* : "this" (the attribute) is predicated on "that" (the substance) in "this way" (the mode).

Consider for example the phrase "a student learns mathematics"
*
(y.a.-' b.-j.-s.y.-' E:E:T:.-'
+
y.a.-' d.a.-s.a.-f.o.-' E:E:S:.-')
**

Terms

E:E:S:.	nominative (subject)
E:E:T:.	accusative (object complement)
y.a.-	learn
b.-j.-s.y.-'	mathematics
d.a.-s.a.-f.o.-'	student

1. The first clause (*y.a.-' b.-j.-s.y.-' E:E:T:.-'*) says that "mathematics" are predicated on the verb "learn" on the mode of "the object".

2. The second clause (*y.a.-' d.a.-s.a.-f.o.-' E:E:S:.-'**) says that "the student" is predicated on the verb "learn" on the mode of the "subject".

The preceding example illustrates the three roles *in the syntagmatic circuits*. Now let us consider the three roles *in the inter-textual circuits*. Earlier in our discussion of hypertexts (see 3.2.1.2, point 4) we saw that, above the level of the text, the three roles are interpreted according to the schema: "text to which one refers" (substance) / "text that makes reference" (attribute) / "note of reference" (mode).

In the syntagmatic circuits as well as in the inter-textual circuits, the three modes construct *an auto-referential predictive loop* in which the signification of each of the units that plays one of the roles depends on the signification of the units that play the two other roles.

Ultimately, the three roles trace back to the triplets (arrival vertex, departure vertex, label of edge) of graph theory. This is precisely why the semantic operations of IEML can be translated automatically into the form of graphs. A multiplication can be read as the construction of an edge between two vertices and an addition of multiplications as the construction of a *set* of edges between vertices, in other words like the construction of a graph. At a higher layer or degree[10], agraph itself becomes a vertex that will be connected to other vertices by new edges.

3.2.3.2 The substance role

Substance is that on which something else is predicated, or "the subject" of the predication in the Aristotelian sense of the term. It determines (1) that about which one speaks and (2) the grammatical orientation of an expression. In the inter-textual circuits, substance role designates the text cited or in reference.

If the first character, or the*initial*, of a substance is of the type *O:** then the substance is a verb or the verbal root of a word and the whole sequence itself is a verb or a phrasal verb. If the initial of a substantive unit is of the type *M:**, then the substance is a noun, the nominal root of a noun or of an adjective, and the totality of the category is itself a noun or a nominal phrase. Finally, we have seen that the adverbs, preposition, conjunctions, pronouns and various inflections, such as cases and conjugations, begin with *E:**. The initial of their substance is "empty".

Thus it is clear that the notion of substance, in the sense that it has in the technical vocabulary of IEML, must not be interpreted as excluding the notions of quality, of process, of event, of manner, or of relation. The substance role of the predication can be played by a unit of meaning that belongs to whichever of the three classes.

3.2.3.3 The role of attribute

The attribute is a quality, a distinctive trait, an actualizing factor (such as the subject of a verb) or a complement of the substance. In the inter-textual circuits, the role of attribute designates *the citing text* that constructs the text cited as a reference.

[10]The layer for syntagmatic circuits and the degree for inter-textual circuits

In terms of the dictionary, the substance indicates the main notion of the semantics of a term, whereas the attribute indicates a variation on the principal notion or a supplementary property.

In the verbal phrases, the attribute can be the subject, the object or any sort of complement of the verb that plays the substantive role. The precise grammatical function of the attribute (i.e., the relation between the substance and the attribute) is indicated by the mode.

In the nominal phrases, the attribute can be the "attribute" of the noun (in the ordinary grammatical sense of the term), a genitive or any complement of the noun that plays the substantive role. Once again, the precise grammatical function of the attribute is noted by the mode.

3.2.3.4 The role of mode

*In terms of the dictiona*ry, the mode is used to indicate a variant of the signification of a terminological root, the root being constituted by the substance-attribute group.

In the construction of words, the mode is used to mark the inflections, such as gender, number, person (first, second, third), the modes and the tenses of verbs, the case of nouns, etc.

In the construction of phrases (knowing that phrases of phrases, etc., exist), the mode is used to define the grammatical *relation* between the unit in substance role and the unit in attribute role.

In 2012, the IEML dictionary already contained more than 150 grammatical cases, conjugations, pronouns, adverbs and prepositions[11] from layers 1, 2 and 3. These auxiliary terms (which are neither nouns nor verbs) were primariy conceived *to be used in modal role.* Nonetheless, I remind the reader that there is nothing to prevent nominal or verbal units from being used in the role of mode, or auxiliary terms in the role of substance or attribute.

In the inter-textual circuits, the role of mode designates *the note of citation or of reference* that connects the citing text to the cited text (in reference).

3.2.4 From morphemes to propositions

3.2.4.1 IEML Morphemes

A morpheme is a term (or an addition of terms promoted to the same layer), that appear in a syntagmatic circuit.

For example, in :

y.-E:.-(E:S:.O:M:.- + E:S:.wo.-)'

which signifies "I will know", *y.** and *(E:S:.O:M:.- + E:S:.wo.-)** constitute morphemes.

Terms

[11]Post-position would be more appropriate, because the mode comes *after* that which it modifies.

E:S:.O:M:.- future

E:S:.wo.- first person singular (I)

y. to know

3.2.4.2 IEML Words

In IEML, two main types of words exist : word morphemes and inflected words.

Word-morphemes Word-morphemes or non-inflected words *do not carry* any case, gender, number, conjugation, articles, prepositions, etc. For example, *y.** "to know" or *(y. + e.)* "to know and to be able" are word-morphemes.

Inflected words In inflected words, the root morpheme is placed in the role of substance, the morpheme of flexion is placed in the role of mode and the attribute role is occupied by an empty series.
First example:
y.-E:.-(E:U:A:.- + E:S:.O:M:.- + E:T:.wo.-)'
signifies : "They will not know." and constitutes one word.

Terms

E:U:A:. negation

y. to know

E:S:.O:M:.- future

E:T:.wo.- third person plural (they)

Second example:
*m.i.-k.i.-'E:.-'E:U:.wa.-',**
signifies: "these games" and constitutes one word.

Terms

m.i.-k.i.-' game

E:U:.wa.- plural demonstrative pronoun

3.2.4.3 IEML Clauses

Degree-1 clauses are multiplications of additions of words in which the unit in the role of mode explicates the grammatical relation that the unit in attribute role holds with the unit in substance role.

For example,
*
(u.-E:.-(E:B:.y.- + E:T:.wo.-)' b.a.-'E:E:S:.-',
+

u.-E:.-(E:B:.y.- + E:T:.wo.-)' t.o.-b.o.-'E:E:T:.-',)
**
is a phrase that can be analyzed in two clauses :

1. The first clause contains three words: "a storyteller" *b.a.-** "considered as subject" *E:E:S:.-** "tells" *u.-E:-(E:B:.y.-+ E:T:.wo.-)'**

2. The second clause contains three words: "tells" *u.-E:-(E:B:.y.-+ E:T:.wo.-)'**, "a story" *t.o.-b.o.-'** "considered as object" *E:E:T:.-**

Thus, the whole phrase together signifies: "A storyteller tells a story".

Terms

E:E:S:.	nominative (subject)
E:E:T:.	accusative (object complement)
u.	state (translates "tell" in the phrase because the object is a story)
E:B:.y.-	present of the indicative
E:T:.wo.-	third person singular
b.a.-	storyteller
t.o.-b.o.-'	story, account, tale

3.2.4.4 IEML Phrases

Phrases of level 1 are additions of degree-1 clauses, degree-2 clauses are multiplications of degree-1 phrases, degree-2 phrases are additions of degree-2 clauses and so on.

Example 1 "The child of my neighbour [My neighbour's child] is often authorized by a doctor not to go to school"

This phrase contains four clauses from layer L5, obviously in sequence according to the order of the script.

*
(u.A:.- E:.- (E:U:A:.- + E:U:.e.- + E:B:T:.-)', h.y.-', E:T:.f.-',_
+
u.A:.- E:.- (E:U:A:.- + E:U:.e.- + E:B:T:.-)', n.a.-f.a.-f.o.-', E:B:.x.-',_
+
u.A:.- E:.- (E:U:A:.- + E:U:.e.- + E:B:T:.-)', d.a.-m.a.-f.o.-' E:.-' E:A:.p.-', E:E:S:.-',_
+
d.a.-m.a.-f.o.-' E:.-' E:A:.p.-', m.a.-k.a.-f.o.-' E:.-' (E:A:.wo.-' + E:S:.wo.-' + E:F:.we.-'), E:E:A:.-', _)
**

Terms

E:E:A:.	genitive
E:E:S:.	nominative
E:U:A:.	negation
E:B:T:.	often
E:U:.e.-	passive habilitative mode
E:A:.wo.-	possessive singular
E:A:.p.-	definite article
E:S:.wo.-	I, me (first person singular)
E:B:.x.-	agent
E:T:.f.-	in (enter into)
u.A:.-	to go
h.y.-	school
E:F:.we.-	feminine gender
m.a.-k.a.-f.o.-'	neighbour
n.a.-f.a.-f.o.-'	doctor [physician]
d.a.-m.a.-f.o.-'	child

<u>Translation of the phrase clause by clause</u>

1. Let us begin as follows :
   u.A:.- E:.- (E:U:A:.- + E:U:.e.- + E:B:T:.-)', h.y.-', E:T:.f.-',_
   u.A:.- E:.- (E:U:A:.- + E:U:.e.- + E:B:T:.-)' signifies "often authorized (ability) not to go" where "to go" *u.A:.-** is the root of the verb and *(E:U:A:.- + E:U:.e.- + E:B:T:.-)** are its three inflections. *****h.y.-** is the school and *****E:T:.f.-** shows that the school is the complement of place of the verb.

2. **u.A:.- E:.- (E:U:A:.- + E:U:.e.- + E:B:T:.-)', n.a.-f.a.-f.o.-', E:B:.x.-',_**
   Here the verb in the passive mode is the same as in the preceding clause and *n.a.-f.a.-f.o.-',** (the doctor) is indicated as agent complement *E:B:.x.-',**.

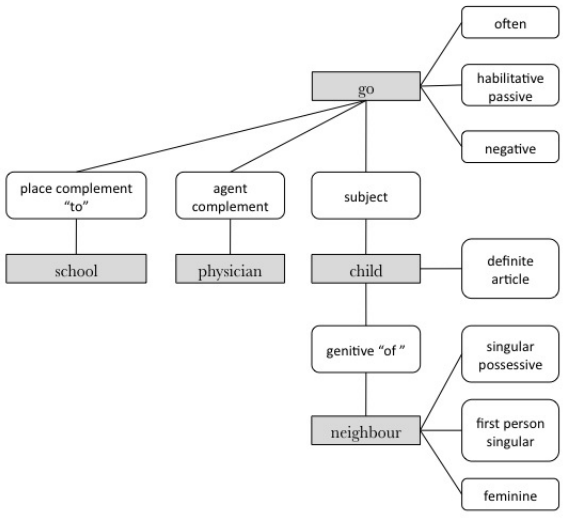


Figure 3.9: My [female] neighbour's child is often authorized by a doctor not to go to school

3. **u.A:.- E:.- (E:U:A:.- + E:U:.e.- + E:B:T:.-)', d.a.-m.a.-f.o.-' E:.-' E:A:.p.-', E:E:S:.-',_**
   In this third clause the verb again is the same, but it is the subject *E:E:S:.** that is indicated, namely the child (root *d.a.-m.a.-f.o.-'** and flexion *E:A:.p.-** to mark the definite article).

4. **d.a.-m.a.-f.o.-' E:.-' E:A:.p.-', m.a.-k.a.-f.o.-' E:.-' (E:A:.wo.-' + E:S:.wo.-' + E:F:.we.-'), E:E:A:.-', _**
   In this fourth and final clause, the substance is played by "the child" *d.a.-m.a.-f.o.-' E:.-' E:A:.p.-',** and the child in question is qualified : it is [the child] of (genitive *E:E:A:.-**) "my neighbour" *m.a.-k.a.-f.o.-' E:.-' (E:A:.wo.-' + E:S:.wo.-' + E:F:.we.-'),**. The flexion *(E:A:.wo.-' + E:S:.wo.-' + E:F:.we.-')** simultaneously marks the possessive singular, the first person singular and the feminine, thus my [*female*] neighbour.

Semes in roles of substance and of attribute of clauses from the same phrase are considered as the vertices of a graph. A seme in the role of mode is considered as the label of the edge connecting the vertex-seme in substance role and the vertex-seme in attribute role. This label indicates the grammatic relation between the two vertices. Thus it is that the phrase we have just analyzed, as all IEML phrases, can be represented as a *semantic network,* as can be seen in figure 3.9, where the verbs and the nouns are represented against a grey backdrop, and the auxiliaries against a white backdrop.

Example 2 In this second example, I will illustrate the case of two level-1 phrases (the main proposition and the subordinate proposition) connected by a phrase in the role of mode : “I learned mathematics because I followed a good professor”. Thus, the whole of the proposition is a level-2 clause.

```
*
y.a.- E:.- (E:S:.wo.- + E:T:.e.-)’ b.-j.-s.y.-’ E:E:T:.-’,_
a.o.- E:.- (E:S:.wo.- + E:T:.o.-)’, d.a.-f.a.-f.o.-’ E:.-’ E:U:T:.-’, E:E:T:.-’,_
E:M:.B:O:.-’,_;
**
```

Terms

E:E:T:. accusative (direct object complement)

E:U:T:. good, better

E:S:.wo.- I (first person singular)

E:T:.o.- past optative

E:T:.e.- past habilitative

a.o.- to follow

y.a.- to learn

E:M:.B:O:.- cause, functional role “because”

b.-j.-s.y.-’ mathematics

d.a.-f.a.-f.o.-’ teacher

1. The substance phrase or, if you will, the main proposition signifies “I learned mathematics”. The mode of the verb indicates an habilitative nuance (I was *able to* learn).

2. The attribute phrase or, if you will, the subordinate proposition signifies “I followed a good teacher”. The mode of the verb indicates an optative nuance (I *wanted to* follow). Note here that “good” *E:U:T:.** acts as an inflection of the root “teacher” *d.a.-f.a.-f.o.-’**.

3. The mode phrase (this is clearly a phrase-word here) indicates the relation “because” between the substance phrase and the attribute phrase.

The semantic network that corresponds to this level-2 clause is illustrated in the figure3.10, where the nouns and the verbs appear against a grey backdrop and the auxiliaries against a white backdrop.

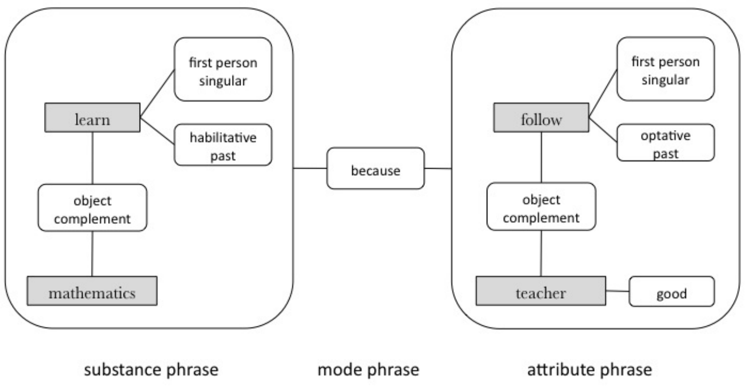


Figure 3.10: I learned mathematics because I followed a good professor

3.2.5 From propositions to texts (USL)

3.2.5.1 Strategy for writing USLs

It goes without saying that, for those end-users who do not wish to concern themselves with fabricating USLs, it will always be possible to use graphic interfaces that will allow them to formulate simple tags in natural languages, these tags being automatically translated into IEML.

Remember first of all that every USL obviously represents *a possible interpretation* of the indexed object, document or set of data : the interpretation of an individual or of a creative conversation. Thus, people will be able to invent several distinct USLs in order to categorize one given document, according to their points of view and their universes of discourse. Moreover, there is nothing to prevent a USL from being re-used in order to remove or add propositions as a way of personalizing the categorization.

The general strategy for constructing a USL consists of describing the "semantic spectrum" that gives an account of a document. Above all, this is not about seeking an equivalent of the complex concept to be described in IEML in *a single term* of the dictionary. The more vast and precise the semantic spectrum – expressed by *several* units of meaning from *several* layers – the better will function the operations of selection (difference, inclusion, union...), the connections with other USLs, and the calculation of semantic distances.

A USL covers a number of layers. The lowest layers designate the most general semantic components, whereas the highest layers represent the most specific or most precise semantic components.

3.2.5.2 The example of “Wikipedia”

Conventions of writing

In the writing used for USLs, distinct propositions are always separated by a slash. To simplify reading, I have separated the *catsets* from layers L0, L1, L2, and L3 by a slash preceded an followed by a line return. *Propositions* from layers L4 and L5 likewise are separated by a slash and followed by a line return. The distinct clauses of a same phrase are joined by a plus sign + preceded and followed by a line return. I will use the same conventions in the following examples. The USL here below is an example of an IEML text describing *Wikipedia*.

```
*
(U:+S:)
/
t. / d.
/
wo.y.- / wa.k.- / e.y.- / s.y.- / k.h.-
/
a.u.-we.h.-’ / n.-y.-s.y.-’
/
(n.-y.-s.y.-’ s.o.-k.o.-’ E:E:A:.-’,
+
n.-y.-s.y.-’ b.i.-b.i.-’ E:A:.m.-’,)
/
u.e.-we.h.-’ m.a.-n.a.-f.o.-’ E:E:S:.-’,
/
(e.-’ (b.a.-b.a.-f.o.-’+ t.a.-b.a.-f.o.-’) E:E:S:.-’,_
+
e.-’ s.e.-k.u.-’ E:E:T:.-’,_
+
(b.a.-b.a.-f.o.-’+ t.a.-b.a.-f.o.-’) (E:F:.wa.-’ + E:A:T:.-’) E:E:U:.-’,_
+
(b.a.-b.a.-f.o.-’ + t.a.-b.a.-f.o.-’) l.o.-m.o.-s.u.-’ E:E:U:.-’,_
+
s.e.-k.u.-’E:.-’E:B:.s.-’, k.i.-b.i.-t.u.-’, E:T:.x.-’,_
+
s.e.-k.u.-’, b.-u.-’E:.-’E:F:.wa.-’, E:T:.x.-’,_
+
k.i.-b.i.-t.u.-’ b.x.-’ E:E:U:.-’,_)
**
```

Translation of the USL, in the order of the script

L0 networks of knowledge

L1 memory / truth

L2 to be oriented within knowledge / act in the common good / synthesize / organized knowledge / collective creation

L3 open the public space / encyclopedia

L4 encyclopedia of collective intelligence in cyberspace / volunteers producing didactic material / authorizes open, plural writing and edition in several languages, by means of a collaborative software environment.

All the catsets from layers L0 to L3 are composed of terms from the dictionary for which the english descriptor is reproduced in the translation above. I will now explain the translation of the three propositions from the L4 catset.

First proposition of layer L4
*(n.-y.-s.y.-' s.o.-k.o.-' E:E:A:.-',
+
n.-y.-s.y.-' b.i.-b.i.-' E:A:.m.-',)**

Terms

n.-y.-s.y.-'	encyclopedia
s.o.-k.o.-'	collective intelligence
b.i.-b.i.-'	digital medium (or cyberspace)
E:E:A:.	genitive (noun complement)
E:A:.m.-	in, within, in the middle of

This proposition is a phrase composed of two clauses from layer L4 in a relation of addition. The clauses are nominal because the two sequences begin with a consonant. In the two clauses, the auxiliary mode from layer 3 indicates the relation between the substance unit from layer L3 and the attribute unit from L3.

1. The first clause signifies : "encyclopedia of collective intelligence"
2. The second clause signifies : "encyclopedia in the digital medium"

Thus we obtain (automatically) the diagram seen in figure 3.11, which translates into english as "encyclopedia of collective intelligence in the digital medium". In this diagram, as in the preceding ones, the verbs and nouns are on a grey background and the auxiliaries are on a white background.

Second proposition from layer L4
*u.e.-we.h.-' m.a.-n.a.-f.o.-' E:E:S:.-',**

Terms

u.e.-we.h.-'	to produce didactic material
m.a.-n.a.-f.o.-'	volunteer

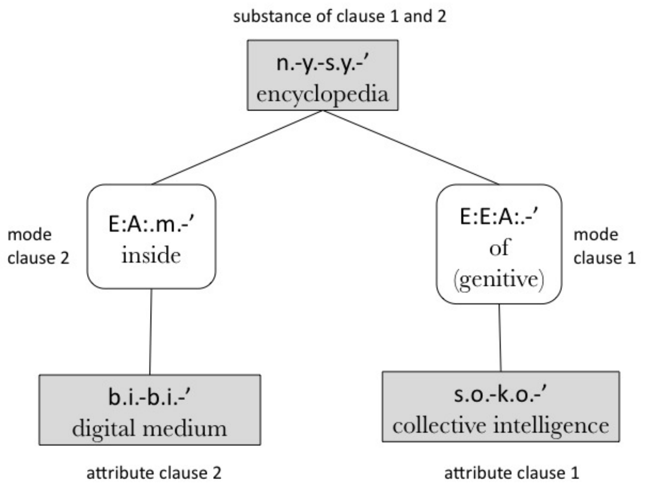


Figure 3.11: Encyclopedia of collective intelligence in the digital medium

E:E:S:.- nominative (marking the relation between verb and subject)

Note that here there is no indication of number. The fact that there is no plural does not indicate the singular, but rather that we wish to designate a concept in general. In the same manner, the absence of time, of mode, and of person to modify the verb indicates that what is to be understood is the action of producing didactic material in general. The signification of this verbal phrase composed of a single clause is thus "volunteers producing didactic material". Many natural languages require choosing between singular and plural, or using a definite or indefinite article, but this is not the case with IEML.

Proposition from layer L5

Terms

E:E:U:. noun attribute

E:E:S:. nominative

E:E:T:. direct object complement

E:A:T:. much, very

e. to be able

E:B:.s.- in an open space

E:T:.x.- auxiliary of means

b.x.- collaborative environment

E:F:.wa.- auxiliary marking the plural

s.e.-k.u.-' competence in reading and writing

b.-u.-' natural language

b.a.-b.a.-f.o.-' author

t.a.-b.a.-f.o.-' editor

k.i.-b.i.-t.u.-' software

l.o.-m.o.-s.u.-' free

```
*
   (e.-' (b.a.-b.a.-f.o.-'+ t.a.-b.a.-f.o.-') E:E:S:.-',_
   +
   e.-' s.e.-k.u.-' E:E:T:.-',_
   +
   (b.a.-b.a.-f.o.-'+ t.a.-b.a.-f.o.-') (E:F:.wa.-' + E:A:T:.-') E:E:U:.-',_
   +
   (b.a.-b.a.-f.o.-' + t.a.-b.a.-f.o.-') l.o.-m.o.-s.u.-' E:E:U:.-',_
   +
   s.e.-k.u.-'E:.-'E:B:.s.-', k.i.-b.i.-t.u.-', E:T:.x.-',_
   +
   s.e.-k.u.-', b.-u.-'E:.-'E:F:.wa.-', E:T:.x.-',_
   +
   k.i.-b.i.-t.u.-' b.x.-' E:E:U:.-',_)
   **
```

The preceding IEML phrase is at layer L5 and is composed of seven clauses from layer L5 in a relation of addition.

1. The first clause signifies that the subject of "to be able" are the readers and the editors.
2. The second clause signifies that the object of the verb "to be able" is the competence of reading-writing.
3. The third clause highlights the multiplicity of the readers and editors.
4. The fourth clause affirms the liberty of these same readers and editors.
5. The fifth clause indicates that the "open" reading-writing is done by means of software.
6. The sixth clause indicates that the competence in reading-writing is practiced via several languages.

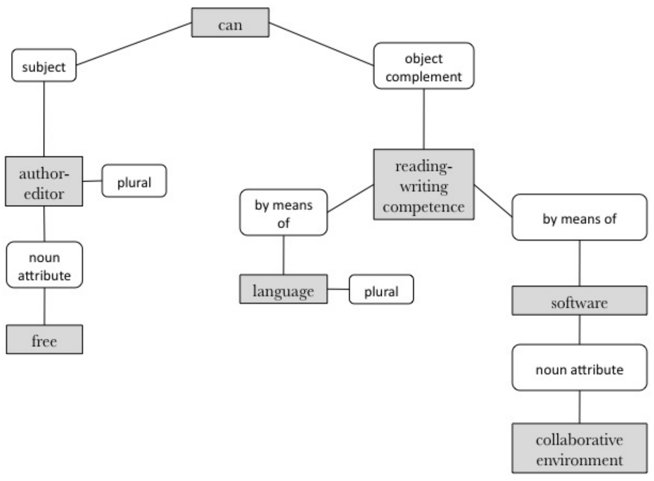


Figure 3.12: Syntagmatic network of an IEML phrase with seven clauses

7. The seventh clause affirms that the software has the property of being a collaborative environment.

Reading or writing this whole phrase comes down to conceiving the semantic network, as in figure 3.12. To simplify the visual representation, I do not show the IEML text. The nouns and verbs are on a grey background, while the auxiliaries are on a white background. This figure 3.12 gives an idea of the possible appearance of the semantic network in natural language produced automatically from a phrase in IEML (with the language being chosen by the user).

3.2.5.3 Example of set operations on USLs

It is possible to carry out set operations on USLs. Imagine two USLs: “XML” and “Web_of_data”

“XML” *
(A:+S:)
/
b.
/
we.b.- / we.g.-
/
e.o.-we.h.-’ / b.i.-b.i.-’ / t.e.-d.u.-’
/

(i.i.-we.h.-', b.-j.-'E:.-'E:A:.g.-', E:T:.x.-',_
+
i.i.-we.h.-' s.a.-t.a.-' E:E:T:.-',_)
**

L0 networks of documents

L1 language

L2 cultivate systems of information / unify the documentation

L3 establish norms and standards / digital medium / respond to needs for information

L5 guarantee the compatibility of the data thanks to the same formal structure.

All the catsets from layers 0 to 3 are composed of terms from the dictionary and whose english descriptor is reproduced literally in the translation here above (the descriptors appear in the translation exactly in the order of the corresponding IEML propositions). I will now explain the translation of the category from layer 5.

Terms

i.i.-we.h.-'	to ensure compatibility
b.-j.-'	formal structure
s.a.-t.a.-'	data
E:E:T:.	accusative (direct object complement)
E:A:.g.-	as, like, same
E:T:.x.-	by means of

1. The first verbal clause connects the verb in substance *i.i.-we.h.-',** "ensure compatibility" to the noun in attribute *b.-j.-'E:.-'E:A:.g.-',** by the auxiliary in role of mode *E:T:.x.-,** marking the means.
   *b.-j.-'E:.-'E:A:.g.-',** is itself a noun composed of two morpheme-terms :
   - the nominal term *b.-j.-'**, which signifies "formal structure"
   - the auxiliary term *E:A:.g.-** modifying the noun and which signifies "same, like, as".
   *b.-j.-'E:.-'E:A:.g.-',** thus signifies "same formal structure"

2. The second verbal clause signifies: ensure the compatibility (the verb in substance *i.i.-we.h.-'**) of the data (the noun in attribute *s.a.-t.a.-'**). The relation between the verb in substance and the noun in attribute is described by the word auxiliary in the role of mode *E:E:T:.-'** marking the direct object complement.

The complete phrase gives “To guarantee the compatibility of data thanks to (by means of) the same formal structure.”

Note that the terms *s.a.-t.a.-’** (data) and *s.a.-t.a.-f.o.-’** (information specialist) have the same substance and the same attribute, *s.a.-t.a.-f.o.-’** being the specialist who organizes and manipulates *s.a.-t.a.-’**. This also holds for *b.-j.-’** (formal structure) and *b.-j.-s.y.-’** (mathematics), the one being the object of the other.

I will now give the example of a second USL, which I will subsequently unite and intersect with the preceding.

“Web_of_data” *
(U:+S:) / (A:+S:)
/
d.
/ s.x.- / x.j.-
/
e.o.-we.h.-’ / b.i.-b.i.-’
/
e.o.-we.h.-’ b.i.-b.i.-’ E:E:B:.-’,
/
s.a.-t.a.-’ b.i.-l.i.-t.u.-’ E:E:U:.-’,
/
(t.e.-t.u.-wa.e.-’ k.i.-b.i.-t.u.-’ E:E:A:.-’,
+
k.i.-b.i.-t.u.-’ b.e.-’ E:A:.x.-’,)
**

L0 networks of knowledge / networks of documents

L1 truth

L2 technical project / evolution of computing

L3 establish norms and standards / digital medium

L4 establish norms and standards for the digital medium / linked data / conception of software robots with capacities for reasoning

I will now explicate the translation of the last two propositions from layer L4, starting with the translation of the terms.

Terms

E:E:U:.	noun attribute
E:E:A:.	genitive
E:E:B:.	dative (to the benefit of)

E:A:.x.-	with
b.e.-	capacity for reasoning
s.a.-t.a.-'	data
b.i.-l.i.-t.u.-'	linked
t.e.-t.u.-wa.e.-'	competence for software design
k.i.-b.i.-t.u.-'	software

1. The second proposition form layer L4 is a nominal phrase composed of a sole clause : the data *s.a.-t.a.-'** are (noun attribute *E:E:U:.**) linked *b.i.-l.i.-t.u.-'**, which translates the concept of "linked data".

2. The third proposition from layer L4 is a phrase with two clauses. The first clause indicates a competence for the design *t.e.-t.u.-wa.e.-'** of (*E:E:A:.** genitive) software *k.i.-b.i.-t.u.-'** and the second clause indicates that the software has a capacity for reasoning (with: *E:A:.x.-**, capacity for reasoning: *b.e.-**).

Results of the two operations on the preceding USLs Set operations on USLs are always carried out on sets of propositions (and thus on sets of sequences) *layer by layer.*

The *intersection* of the two preceding USLs gives a new USL (that one could call "Internet_standards") containing a proposition from layer 0 and two propositions from layer 3.

*
(A:+S:)
/
e.o.-we.h.-' / b.i.-b.i.-'
**

L0	networks of documents
L3	establish norms and standards / digital medium

The *union* of the two USLs gives the USL that one could call the "Great_Web_of_data"

*
(U:+S:) / (A:+S:)
/
b. / d.
/
we.g.- / we.b.- / s.x.- / x.j.-
/
e.o.-we.h.-' / b.i.-b.i.-' / t.e.-d.u.-'
/
e.o.-we.h.-' b.i.-b.i.-' E:E:B:.-',

```
/
s.a.-t.a.-' b.i.-l.i.-t.u.-' E:E:U:.-',
/
(t.e.-t.u.-wa.e.-' k.i.-b.i.-t.u.-' E:E:A:.-',
+
k.i.-b.i.-t.u.-' b.e.-' E:A:.x.-',)
/
(i.i.-we.h.-', b.-j.-'E:.-'E:A:.g.-', E:T:.x.-',_
+
i.i.-we.h.-' s.a.-t.a.-' E:E:T:.-',_)
**
```

L0: networks of knowledge / networks of documents

L1: language / truth

L2: unify the documentation / cultivate systems of information / technical project / evolution of computing

L3: establish norms and standards / digital medium / respond to needs for information

L4: establish norms and standards for the digital medium / *linked data*/ conception of software robots with capacities of reasoning

L5: guarantee compatibility of data thanks to a same formal structure

3.2.6 Representation of the syntactic relations between units of enunciation

In IEML two main types of semantic circuits exist: paradigmatic circuits, which we examined in chapter 3.1, and enunciative circuits, the main linguistic aspects of which we have just seen in the preceding sections. Recall what we already know about enunciative circuits : they are composed of a complex hierarchy of propositions, of texts (additions of propositions), and of hypertexts (additions and multiplications of texts). Propositions are themselves constructed from a complex hierarchy of morphemes, of words, of clauses, of phrases and of phrases of phrases. At the base of this hierarchy of units of meaning are the terms, which are defined by the keys of the dictionary (see 3.2.1.2).

In IEML there exists a strict parallelism between the syntactic relations that connect the units of meaning, on the one hand, and the channels that connect the reservoirs of the enunciative circuits, on the other hand. This parallelism is mediated by the operations of addition and of multiplication. Recall that these operations command the relations of order and of symmetry between their variables in a rather simple manner :

1. for addition, a relation of order links each of the input variables to the output variable, and a relation of symmetry links the input variables of the same operation ;

CANAUX	**Multiplication** $\otimes$	**Addition** $\oplus$
Order $>$	- word $\otimes >$ morpheme - clause $\otimes >$ word - clause of phrases $\otimes >$ phrase - hyp. clause $\otimes >$ texte	- morpheme $\oplus >$ term - phrase $\oplus >$ clause - text $\oplus >$ proposition - hypertext $\oplus >$ hyp. clause
Symmetry $\sqcap$	- word $\otimes\sqcap$ inverted word - clause $\otimes\sqcap$ inverted clause - cl. of ph. $\otimes\sqcap$ cl. of inverted ph. - hyp. clause $\otimes\sqcap$ inverted hyp. cl.	- term $\oplus\sqcap$ term - word $\oplus\sqcap$ word - clause $\oplus\sqcap$ clause - proposition $\oplus\sqcap$ proposition - hyp. clause $\oplus\sqcap$ hyp. clause

Table 3.6: The 4 types of channels of enunciative circuits

2. for multiplication, a relation of order links each of the three input variables to the output variable, and a relation of symmetry links the output variables of which two input variables exchanged roles.

Table3.6 reviews the representation of the syntactic relations between units of meaning via channels between reservoirs. I will now comment briefly on the four types of channels presented in the table.

3.2.6.1 Channels of multiplicative order

The channels of multiplicative order connect units of meaning *from level*[12] *n+1* which comprises them. For example, morphemes and the word that contains them, words and the cause that articulates them, and so on. Here, remember that a text (USL) and the propositions that it contains *are not* connected by channels of multiplicative order, but by channels of additive order.

3.2.6.2 Channels of additive order

Channels of additive order connect units of meaning from level n to a unit of meaning *from the same level*[13] *n* that contains them. For example : terms and the morpheme that contains them, clauses and the phrase that contains them, etc.

3.2.6.3 Channels of multiplicative symmetry

The channels of multiplicative symmetry connect two units of the same level of which two semes inverse their roles and of which the third seme is identical. For example, the phrase "The history of the motor is an agent of the motor of history" is translated as follows into IEML :

[12]By level I understand here the layer, the degree, the order, etc.
[13]See the preceding note.

*k.o.-t.o.-'E:.-'E:A:.p.-',n.i.-s.i.-'E:.-'E:A:.p.-',E:E:A:.-',_n.i.-s.i.-'E:.-'E:A:.p.-',k.o.-t.o.-'E:.-'E:A:.p.-',E:E:A:.-',E:B:.x.-', _;**

In this case, the two clauses

*k.o.-t.o.-'E:.-'E:A:.p.-',n.i.-s.i.-'E:.-'E:A:.p.-',E:E:A:.-',_**

and

*n.i.-s.i.-'E:.-'E:A:.p.-',k.o.-t.o.-'E:.-'E:A:.p.-',E:E:A:.-',**

are in a relation of multiplicative symmetry.

Terms

E:E:A:.	genitive
E:A:.p.-	definite article
E:B:.x.-	agent, effective cause
k.o.-t.o.-'	history
n.i.-s.i.-'	motor

At the inter-textual level, we will find channels of multiplicative symmetry between USLs that cite each other.

3.2.6.4 Channels of additive symmetry

The channels of additive symmetry interconnect in a complete graph the units of level n that work towards the construction of an encompassing unit from the same level n. For example, the terms of the same morpheme, the clauses of the same phrase, the propositions of the same text, etc.

3.2.7 Summary of the hierarchy of units of meaning

Figure 3.13 summarizes the hierarchy of units of meaning in IEML. This hierarchy is “tangled”: on the one hand the meaning of higher units is conditioned by the meaning of lower units but, on the other hand, the meaning of lower units is reciprocally conditioned by that of their higher units, which form, degree by degree, their respective contexts.

3.2.8 Correct and incorrect constructions

3.2.8.1 Coding

In order to concisely represent the grammatical patterns according to which the units of meaning are arranged, I will code them as follows :

V represents an empty sequence

Z represents a term

R represents a morpheme

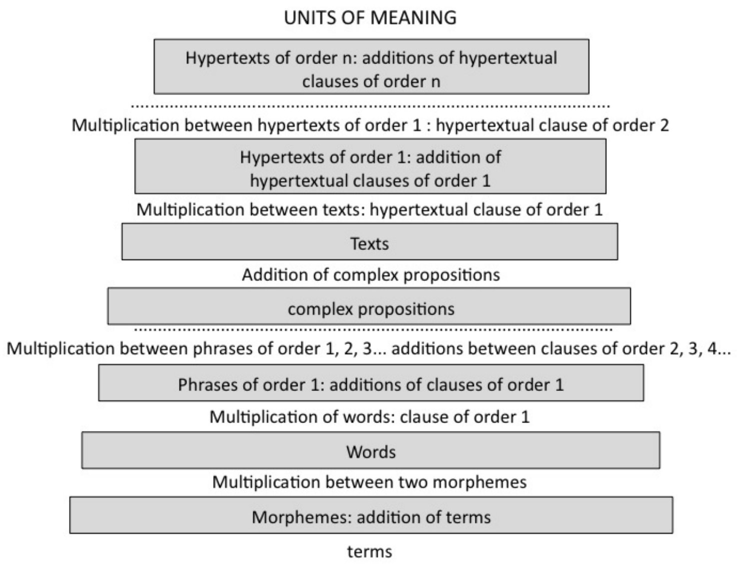

UNITS OF MEANING
Hypertexts of order n: additions of hypertextual clauses of order n
Multiplication between hypertexts of order 1 : hypertextual clause of order 2
Hypertexts of order 1: addition of hypertextual clauses of order 1
Multiplication between texts: hypertextual clause of order 1
Texts
Addition of complex propositions
complex propositions
Multiplication between phrases of order 1, 2, 3... additions between clauses of order 2, 3, 4...
Phrases of order 1: additions of clauses of order 1
Multiplication of words: clause of order 1
Words
Multiplication between two morphemes
Morphemes: addition of terms
terms

Figure 3.13: The tangled hierarchy of units of meaning

W' represents a word-morpheme

W" represents an inflected word

W represents a word-morpheme or (*exclusive*) an inflected word

C^n represents a clause of order 1, or 2, or 3, etc., with n indicating the order number of the clause

C represents a clause, whatever its order number

P^n represents a phrase of order 1, or 2, or 3, etc., with n indicating the order number of the phrase

P represents a phrase, whatever its order number

Q represents whichever unit of meaning (not entirely empty).

3.2.8.2 Promotion

In every addition $\oplus$ or multiplication $\otimes$ of units of meaning Q = {Z, R, W, C, P} the variables at the lowermost layers are promoted by an empty sequence until reaching the layer of the variable that is at the highest layer. With regard to the subject of promotion, I refer the reader back to section 2.2.6.3 for the syntactic aspect and to section 3.2.1.4 for the semantic aspect.

3.2.8.3 Grammatical constructions

- R = (Z $\oplus$ Z $\oplus$ ...)
- W' = R
- W" = (R $\otimes$ E $\otimes$ R)
- C^1= {(W $\otimes$ W $\otimes$ W), (W' $\otimes$ E $\otimes$ W"), (W" $\otimes$ E $\otimes$ W'), (W" $\otimes$ E $\otimes$ W"), (W $\otimes$ E $\otimes$ E)}
  Note 1 : A construction of the type (W' $\otimes$ E $\otimes$ W') comes down to (R $\otimes$ E $\otimes$ R), which necessarily is a word. Thus, this cannot be a clause.
  Note 2 : clauses of the type (W $\otimes$ E $\otimes$ E) can only be used in phrases of order 1 P^1
- P^1 = (C^1 $\oplus$ C^1 $\oplus$ ...)
- C^2 = {(P^1 $\otimes$ P^1 $\otimes$ P^1), (P^1 $\otimes$ E $\otimes$ P^1), (P^1 $\otimes$ E $\otimes$ E)}
  Note : clauses of the type (P^1 $\otimes$ E $\otimes$ E) can only be used in phrases of order 2 P^2.
- P^2 = (C^2 $\oplus$ C^2 $\oplus$ ...)
- For clauses of an order higher than 2 :
  C^{n+1} = {(P^n $\otimes$ P^n $\otimes$ P^n), (P^n $\otimes$ E $\otimes$ P^n), (P^n $\otimes$ E $\otimes$ E)}
  Note : clauses of the type (P^n $\otimes$ E $\otimes$ E) can only be used in phrases of order n+1 P^{n+1}.
- For phrases of an order higher than 2 :
  P^n = (C^n $\oplus$ C^n $\oplus$ ...)

3.2.8.4 Non-grammatical operations

- An operation of the form (Q $\otimes$ Q $\otimes$ E) is not grammatical, because the relation between substance and attribute would remain non-defined.
- An operation of the form (E $\otimes$ E $\otimes$ Q) is not grammatical, because the mode would have nothing to modify or to connect .
- An operation of the form (E $\otimes$ E $\otimes$ E) is not grammatical, because it signifies nothing, except as an empty series following a series that is not completely empty (see above, under 3.2.8.2)
- Operations of the form (W $\oplus$ W $\oplus$ ...) and (P $\oplus$ P $\oplus$ ...) are not grammatical. Instead, we must use operations of the form (C $\oplus$ C $\oplus$ ...).

3.2.8.5 The empty attribute in operations of the form (Q $\otimes$ E $\otimes$ Q)

The attribute is *always* occupied by an empty series in inflected words and it can be occupied by an empty series in clauses. In these two cases, the mode specifies the meaning of the substance instead of defining the relation between the substance and the attribute.

3.3 Principle of the algorithm for transcoding the Rhizome into the semantic sphere

3.3.1 The enunciative circuits have the same structure as the syntagmatic circuits

If we have an algorithm for constructing syntagmatic circuits (which correspond to propositions), it is clear that we can construct an algorithm to construct enunciative circuits in general (which further include texts and hypertexts). First, the propositions of a same *text* are simply reservoirs interconnected by channels of additive symmetry and which converge via channels of additive order toward the reservoir of the USL that contains them. Nonetheless, I note here that the author of a text can suppress channels of additive symmetry between propositions (in other words, reduce the redundancy of the circuit), provided the final circuit between the propositions is connected. Second, the *hypertexts* arise precisely from the same alternating structure of additions and of multiplications as the syntagmata. Therefore, I will limit my demonstration here to showing that we do have access to an algorithm for the construction of syntagmatic circuits.

3.3.2 The basis of the algorithm for construction of semantic circuits is the algorithm transcoding the Script into the Rhizome

In order to obtain an algorithm for the construction of syntagmatic circuits, one need only use the algorithm for the transcoding of the Script into the Rhizome, described in 2.3, with the following modifications.

3.3.2.1 Change in the nature of the units manipulated

Instead of bulbs and filaments, we will speak of reservoirs representing units of meaning and of channels representing grammatical relations between units of meaning.

3.3.2.2 Reading stops at terms

Reading stops descending toward the leaves of trees of additive and multiplicative order *as soon as it arrives at a term*, which implies adding a "term test " to the algorithm.

3.3.3 Qualification of units of meaning

Once the architecture of the circuit has been completed according to the method that has just been described, the algorithm of enunciation must climb back up the channels of multiplicative and additive order, starting from the terms toward the complete proposition in order to *qualify the different units of meaning* (morphemes, words, clauses of nominal, verbal, or auxiliary type). All this occurs according to the rules described in 3.2.8.

3.3.4 Visualization of semantic networks

3.3.4.1 Visualization of the semantic network of a proposition

Once all the units of meaning of a proposition have been qualified, we can then automatically visualize its semantic network for the human reader. To do so, we need only represent:

- each IEML term by its dictionary equivalent (descriptor) in a selected natural language,

- each morpheme as a complete graph of the terms it contains,

- each inflected word (W") as a connection between a root morpheme (in the role of substance) and a morpheme of flexion (in the role of mode).

- each clause (to its respective degree) as a connection between the unit in the role of substance and the unit in the role of attribute, each connection being labelled by the unit on the role of mode,

- each phrase as the union of the connections represented by its clauses in a relation of addition.

Previously we examined examples of this type of representation. In particular, I refer the reader to the figures 3.12, 3.11, 3.10 and 3.9), in which the units in the role of substance and of attribute are represented in grey, and the units in the role of mode in white.

3.3.4.2 Visualization of texts and hypertexts

By default, a text (USL) shall be represented as a complete graph between the propositions that it contains. As for hypertexts (of successive degrees), they are constructed exactly according to the same principle as are phrases, except that hypertextual clauses of the first order are constructed from texts instead of being constructed from words. Texts in the role of mode are the labels or "notes" of the link from the substance text toward the attribute text.

Chapter 4

Transcoding syntax into semantics

"The algorithm is the idiom of modern science" Bernard Chazelle

4.1 Introduction

This chapter introduces IEML transcoding algorithms. The transcoding of syntax into semantics can be decomposed into three transformations: from Algebraic IEML to Script IEML; from Script IEML to the Rhizome IEML; and finally, from the IEML Rhizome to the *Semantic sphere*. These transformations are presented in sections 4.2, 4.3 and 4.4, respectively.

Some of the algorithms use global variables and functions, which are described in sections 4.1.1, 4.1.2, and 4.1.3.

4.1.1 Global variables

1. The parameter `MinLayer` controls the depth of analysis. Its usage ensures that for a Script expression s at layer L_n, all layers l of s such that $MinLayer \leq l \leq L_n$ are analyzed. Used in algorithm 4.5.

2. The parameter `MultSym` decides whether mutiplicative symmetry relations are created. Used in algorithm 4.5.

3. The parameter `MultOrd` controls which multiplicative order relations are created. This parameter defines three logical variables: `substance`, `attribute`, and `mode`. Relations are created if said logical variables are set to *true*. See section 4.1.3.

4. The parameter `RestrictedScript` is a list of Script expressions. Members of this list are excluded from analysis. Used in algorithm 4.5.

5. The parameter `Terms` is a list of Script expressions. Members of this list are used for different purposes. Used in algorithms 4.5 and 4.6.
6. The data structure `Chars` stores a Script expression s in such a way as to allow iterative replacement of characters of layer 0, modification of said characters, and the construction of a Script expression from these characters. Used in algorithm 4.5.
7. Parameter $\mathtt{E}^{\mathtt{A}}$ is a list of additive order relations. Used in algorithms 4.8 and 4.10.
8. Parameter $\mathtt{E}^{\mathtt{B}}$ is a list of additive symmetry relations. Used in algorithms 4.8 and 4.10.
9. Parameter $\mathtt{E}^{\mathtt{C}}$ is a list of multiplicative order relations. Used in algorithms 4.8, 4.9 and 4.10.
10. Parameter $\mathtt{E}^{\mathtt{D}}$ is a list of multiplicative relations. Used in algorithms 4.8, 4.9 and 4.10.
11. The parameter $\mathtt{G} = (\mathtt{V}, \mathtt{E})$ is a graph that defines a rhizome, where $E = E^A \cup E^B \cup E^C \cup E^D$. Used in algorithms 4.5, 4.6, 4.7, 4.8, 4.9, 4.10 and 4.11.
12. Parameters W' and W'' are lists of Script expressions (words). Used in algorithms 4.8, 4.9 and 4.12.
13. Parameter $\mathtt{C}^{\mathtt{n}}$ is a list of Script expressions of layer n (clauses). Used in algorithms 4.7 4.9, 4.10 and 4.12.
14. Parameter $\mathtt{P}^{\mathtt{n}}$ is a list of Script expressions of layer n (phrases). Used in algorithms 4.7, 4.10 and 4.12.
15. Parameter $\mathtt{H}_{\mathtt{NAT}}$ is a hash table that contains a mapping from Script expressions to corresponding natural language entries. Used in algorithm 4.12.
16. Parameters G_{SEM}, $\bar{G}_{SEM}$ and $\bar{\bar{G}}_{SEM}$ are graphs. Used in algorithm 4.12.

4.1.2 Global functions

1. `IEMLOrder(S)`. Implements ordering rules from section 2.2.7 and returns a list of Script expressions ordered by the number of multiplications in that expression. Expressions with the least number of multiplications are first in the list. Set `S` contains sets of algebraic IEML expressions. Used in algorithm 4.1.
2. `IEMLSeme(s, i)`. Finds and returns the i'th seme of s, where s is an algebraic IEML expression of layer $L > 0$ and i is an integer $1 \leq i \leq 3$. Used in algorithm 4.3.

3. `IEMLReplace(S)`. Implements replacement rules from section 2.2.6.3 and returns an algebraic IEML expression. Set `S` contains sets of algebraic IEML expressions. Used in algorithm 4.4.

4. `Layer(s)`. Finds the highest layer of the given Script expression s, where layer l is $0 \leq l \leq 6$. Used in algorithm 4.5.

5. `Parse(s, construct)`. Finds the specified construct of the given Script expression s at the highest layer of that expression. Three types of constructs are defined: *addition* ($\oplus$); *multiplication* ($\otimes$); and *character* (©). Construct *addition* directs the function to find all sub-scripts that are at the same layer as s and return one or more such sub-scripts. Construct *multiplication* directs the function to find all three semes of s and return exactly three sub-scripts. Construct *character* directs the function to find and return all characters of layer 0 of s. Used in algorithms 4.5 and 4.6.

6. `RoleInterversion(s)`. Computes seme permutations of the given Script s and returns a Script expression. Used in algorithm 4.5.

7. `Expand(c)`. Computes the underlying components of the given character c. Input must be at layer 0. For example, character (*U:+B:*) expands into set {*U:,B:*}, character (*O:*) expands into set {*U:,A:*}, character (*I:*) expands into set {*E:,F:*} and character (*F:*) expands into set {*O:,M:*}. The return value may also be an empty set. Used in algorithm 4.6.

8. `CreateAlgebraic(Chars)`. Creates an algebraic IEML expression from the structure *Chars*. For details of said structure, see section 4.1.1. Used in algorithm 4.6.

9. `Pattern(template)`. Each Script expression can be characterized in terms of its semes. For our purposes, we are only interested in which semes (and sub-semes) represent an empty sequence. For instance, a template value of `SES` will direct the function to verify whether the second seme of any Script expression - and only the second - is an empty sequence, whereas the template value `SES` $\otimes$ `EEE` $\otimes$ `SES` will direct the function to verify whether the first and third semes of any Script expression match the pattern `SES` and whether the second seme is an empty sequence. Used in algorithms 4.8, 4.9 and 4.10.

10. `DetType(s)`. Implements ordering rules from section 2.2.7 and marks the Script s with one of the three types defined in section 3.2.2, namely *nominal*, *verbal* or *auxiliary*. Used in algorithm 4.8.

11. `Clique(s)`. Finds all sub-scripts of Script expression s in additive symmetry relation and returns a complete graph relating those sub-scripts. Used in algorithm 4.12.

12. `Translate(s, hashtable)`. Matches a specific Script s with a value specified by the supplied hash table data structure and returns that value. Used in algorithm 4.12.

4.1.3 Global relation functions

There also exist three functions that have the same input type yet compute different types of relations. The functions are:

1. `AdditionRelations`($\mathtt{s}, \mathtt{S}$). Used in algorithms 4.5 and 4.6.
2. `MultiplicationOrder`($\mathtt{s}, \mathtt{S}$). Used in algorithm 4.5.
3. `MultiplicationSymmetry`($\mathtt{s}, \mathtt{S}$). Used in algorithm 4.5.

The inputs are a Script s (or a set containing exactly one Script expression) and a set S of Script expressions. Return values of the functions are a set of edges and vertices (the relations). In all three of the above functions, relations are not created for any $s_i \in S$ where $s_i \in \mathit{RestrictedScript}$. Moreover, the `AdditionRelations` function creates two types of relations, *additive order* and *additive symmetry*. The `MultiplicationOrder` function depends on the settings of `MultOrd` in order to create appropriate relations, and its input set S must contain exactly one element.

4.2 From the Algebra to the Script

The algorithm 4.1 locates a Script expression from an IEML category. For example, for an input set $\{USE, UBE, UTE, ASE, ABE, ATE\}$, algorithm 4.1 will yield *O:M:.**.

Algorithm 4.1 Algorithm SCRIPT_GENERATOR

```
input  : ($S$) where $S$ is a set of IEML strings
output: Script expression
$K$ ;                                        // sets of sets of strings
begin
    $K \longleftarrow \emptyset$ ;                          // initialize candidate set
    $K \longleftarrow$ SEME_COMPRESSOR($S, K$) ;          // find candidate solutions
    return IEMLOrder($K$)
```

For the purposes of clarity, algorithm 4.1 delegates most of the computation to algorithm 4.2, and its complexity of time and space is therefore proportional to the complexity of algorithm 4.2.

4.2.1 SCRIPT_COMPRESSOR algorithm

The purpose of algorithm 4.2 is to ensure that no more character rewriting can take place. Recall that a sequence {*U:+A:*} is rewritten as *O:*, sequence {*E:+F:*} is rewritten as *I:* and sequence {*O:+M:*} is rewritten as *F:*. Thus, there is a limit on how many rewrites can take place and, accordingly, the complexity of algorithm 4.2 is proportional to $K \cdot C_{SEME\_MATCHER} \cdot C_{SCRIPT\_SOLVER}$, where K

Algorithm 4.2 Algorithm SCRIPT_COMPRESSOR

```
input  : $(S, K)$ where $S$ is a set of IEML strings and $K$ is a set of sets of IEML
         strings
output: Set of sets of IEML strings $K$
6  $C$, $Q$ ;                                          // sets of sets of strings
7  begin
8      $C \longleftarrow$ SEME_MATCHER($S$) ;                // seme combinations
9      $Q \longleftarrow$ SCRIPT_ SOLVER($C, S, \emptyset, \emptyset$) ;    // get all solutions for S
10     if $Q \backslash \{S\} = \emptyset$ then                  // input equals output
11         $K \overset{+}{\longleftarrow} S$ ;                     // add candidate solution
12         return $K$ ;                                 // terminate analysis
13     for $\forall q \in Q$ do                          // continue analysis
14         SCRIPT_COMPRESSOR($q, K$) ;                     // recursive call
15     return $K$ ;                                     // terminate analysis
```

is a scalar that accounts for all list membership checks and $C_{SEME\_MATCHER}$ and C_{SCRIPT_SOLVER} are the complexity of algorithms 4.3 and 4.4, respectively.

Algorithm 4.2 is fairly simple and consists of a terminating condition on line 5 and a recursive call on line 9. The algorithm terminates whenever the input (S) is equal to the calculated values on line 4 (Q), which means that there is no other potential solution to consider.

4.2.2 SEME_MATCHER algorithm

Given an input set of IEML strings at the same layer, algorithm 4.3 finds all distinct groups of strings that share exactly two semes. For example, for an input set $\{USE$, UBE, UTE, ASE, ABE, $ATE\}$, algorithm 4.3 will yield the set in table 4.1.

C
$\{USE, ASE\}$
$\{UBE, ABE\}$
$\{UTE, ATE\}$
$\{ASE, ABE, ATE\}$
$\{USE, UBE, UTE\}$

Table 4.1: Result of algorithm 4.3 for set $\{USE$, UBE, UTE, ASE, ABE, $ATE\}$

Lines 1 and 2 list the temporary variables and their types. Line 4 performs initialization, lines 5 through 19 are the main logic loop, and lines 20 through 23 store and return the result of the algorithm.

Algorithm 4.3 Algorithm SEME_MATCHER

```
input : (S) where S is a set of IEML strings
output: Set of sets of IEML strings C
16 $a_i$, $b_i$ ;                                   // string
17 $D_i$ ;                                          // set of strings
18 begin
19     C ⟵ $\emptyset$ ;                            // initialize C with $\emptyset$
20     for $\forall x \in S$ do
21         for $\forall i \in \{1,2,3\}$ do
22             $D_i \longleftarrow x$ ;                 // initialize $D_i$ with $x$
23             $a_i \longleftarrow$ IEMLSeme($x, i$)
24         for $\forall y \in S \backslash x$ do
25             for $\forall i \in \{1,2,3\}$ do
26                 $b_i \longleftarrow$ IEMLSeme($y, i$)
27             if $a_1 = b_1$ then
28                 if $a_2 = b_2$ then
29                     $D_3 \overset{+}{\longleftarrow} y$ ;          // add $y$ to $D_3$
30                 if $a_3 = b_3$ then
31                     $D_2 \overset{+}{\longleftarrow} y$
32             if $a_2 = b_2$ then
33                 if $a_3 = b_3$ then
34                     $D_1 \overset{+}{\longleftarrow} y$
35         for $\forall i \in \{1,2,3\}$ do
36             if $|D_i| \geq 2$ and $D_i \notin C$ then
37                 $C \overset{+}{\longleftarrow} D_i$ ;        // add $D_i$ to C
38     return $C$
```

Each string is composed of three equal-length substrings (semes), which are compared and grouped by the main logic loop (lines 9 through 19). If a particular group does not contain two or more members, or if it was already found, it is not stored (line 21).

The time and space complexity of algorithm 4.3 is proportional to ln^2 and ln, respectively, where l is the length of the input strings, and n is the size of the input set.

4.2.3 SCRIPT_SOLVER algorithm

Algorithm 4.4 finds all unique expressions that result from considering all the distinct groups obtained from the input set S. For example, for an input set $\{USE, UBE, UTE, ASE, ABE, ATE\}$ and groups in table 4.1, algorithm 4.4

will yield the set in table 4.2.

Q
$\{OTE, OSE, OBE\}$
$\{UME, AME\}$

Table 4.2: Result of algorithm 4.4 for set $\{USE$, UBE, UTE, ASE, ABE, $ATE\}$ and input in table 4.1

Lines 1 and 2 list the temporary variables and their types. Lines 13 through 26 are the main logic loop. Lines 4 through 12 are the termination logic of the recursion: if set C is empty, contains only one element, or if all intersections between any of its elements result in an empty set, then the algorithm terminates. Unused strings in S are added to the potential solution (line 9), and if this potential solution was not already found, it is added to the candidate solution set (line 11). In the main logic loop, the solution space is reduced (lines 14 through 18) by adding all members that do not have semes in common with the potential solution set, and removing them from the set of strings to consider. Lines 19 through 25 set up new variables based on the remaining solution space and make a recursive call to process them on line 26.

The complexity of algorithm 4.4 in time and space is proportional to:

$$l \cdot p \cdot \prod_{k=1}^{k=n/2} (n - 2k + 2) \tag{4.1}$$

where l is the length of input strings, n is the size of the input set C (number of elements in C), and p is the average size of elements in C.

4.3 From the Script to the Rhizome

Algorithm 4.5 creates a rhizome (a graph with labelled edges) from an input Script expression. Note that algorithm 4.5 calls algorithm 4.6, which in turn calls algorithm 4.5. Separating the creation of a rhizome into two algorithms is done only to increase the readability of the entire process. The following sections describe this process in detail.

The complexity of algorithms 4.5 and 4.6 is not analyzed separately, since they form a single process. Both algorithms together perform the following function: the input Script expression is analyzed for additive and multiplicative relations and, as the analysis progresses, a tree of a certain depth and breadth is built. The depth of the tree depends on the number of layers and on the breadth of the average number of additive relations n_A on each particular layer. Moreover, the complexity of algorithm 4.6 depends on the average length of the Script expression l_A, the average number of character expansions e_A and the complexity of algorithm 4.1. Overall, the complexity is proportional to $K \cdot n_A \cdot l_A \cdot e_A \cdot C_{SCRIPT\_GENERATOR}$, where K is a scalar that accounts for

Algorithm 4.4 Algorithm SCRIPT_ SOLVER

input : (C, S, R, Q) where C, R and Q are sets of sets of IEML strings, S is a set of IEML strings

output: Set of sets of IEML strings Q

```
$\overline{S}$ ;                                              // set of strings
$\overline{C}$, $\overline{R}$ ;                              // set of sets of strings
begin
    if $\{\forall c_i, c_j \in C \mid c_i \cap c_j\} = \emptyset$ then      // includes C = $\emptyset$
        for $\forall c \in C$ do
            $R \overset{+}{\longleftarrow} c$ ;                        // add remainder of C
            $S \longleftarrow S \backslash c$ ;                        // remove used string
        for $\forall s \in S$ do
            $R \overset{+}{\longleftarrow} \{s\}$ ;                    // add remainder of S
        if $R \not\subseteq Q$ then
            $Q \overset{+}{\longleftarrow}$ IEMLReplace($R$) ;        // add solution R
        return $Q$
    else
        for $\forall c \in C$ do                             // solution space reduction
            if $\{\forall c_i \in C \backslash \{c\} \mid c \cap c_i\} = \emptyset$ then
                $S \longleftarrow S \backslash c$ ;                    // remove multiplication
                $C \longleftarrow C \backslash \{c\}$ ;                // remove multiplication
                $R \overset{+}{\longleftarrow} c$ ;                    // add multiplication
        for $\forall c \in C$ do                             // remaining multiplications
            $\overline{S} \longleftarrow S \backslash c$ ;             // duplicate S, remove $c$
            $\overline{C} \longleftarrow C \backslash \{c\}$ ;         // duplicate C, remove $c$
            for $\forall c_i \in \overline{C}$ do                // solution space reduction
                if $c \cap c_i \neq \emptyset$ then
                    $\overline{C} \longleftarrow \overline{C} \backslash \{c_i\}$ ;   // remove invalid string
            $\overline{R} \longleftarrow R \cup \{c\}$ ;               // duplicate R, add $c$
        return SCRIPT_ SOLVER($\overline{C}, \overline{S}, \overline{R}, Q$) ;   // recursive call
```

all list membership checks and global functions, and $C_{SCRIPT\_GENERATOR}$ is the complexity of algorithm 4.1.

4.3.1 RHIZOME Algorithm

In order to obtain a rhizome, algorithm 4.5 is used without any special settings (see section 4.1.2). The purpose of this algorithm is to find all additive and multiplicative relations and to identify all strings to be analyzed by algorithm 4.6.

Algorithm 4.5 Algorithm RHIZOME

```
input  : ($s$) where $s$ is a script expression
output: Updated global variable $G = (V, E)$
65 $A, B$ ;                                        // set of script expressions
66 begin
67     if Layer($s$) $<$ MinLayer then
68         return ;                                            // end analysis
69     if $s \in$ Terms $\cup$ RestrictedScript then
70         return ;                                            // end analysis
71     $A \longleftarrow$ Parse($s, \oplus$) ;                     // additive relations
72     if $|A| = 0$ then
73         SCRIPT_DECOMPRESSOR($s$) ;                          // bulb analysis
74         $B \longleftarrow$ Parse($s, \otimes$) ;            // multiplicative relations
75         if $|B| > 0$ then
76             if $B \cap$ RestrictedScript $= \emptyset$ then
77                 $G(V, E) \xleftarrow{+}$ MultiplicationOrder($s, B$)
78             if MultSym then
79                 if $\exists\, v \in G = (V, E) \mid$ v $=$ RoleInterversion($s$) then
80                     $G(V, E) \xleftarrow{+}$ MultiplicationSymmetry($s, \{v\}$)
81             for $b \in B$ do
82                 RHIZOME($b$) ;                      // lower layers analysis
83     else
84         if $A \cap$ RestrictedScript $= \emptyset$ then
85             $G(V, E) \xleftarrow{+}$ AdditionRelations($s, A$)
86         for $a \in A$ do
87             RHIZOME($a$)
```

Lines 3 through 6 are the terminating conditions for the algorithm. The input script is then analyzed to determine its structure (line 7). There are two possibilities for the structure: either there is no additive relation at the script layer (line 8) or there are one or more additive relations (line 19). In the second case, additive relations are added to the rhizome (lines 20 through 21), and a recursive call is made to analyze each additive variable in turn (lines 22 through 23), triggering logic on line 8. Lines 8 through 18 perform multiple tasks, including multiplication relation and bulb analysis. Line 9 triggers algorithm 4.6. The structure of the script is then analyzed for multiplicative relations (line 10). If there is no multiplicative relation, no further analysis takes place and the algorithm terminates. If a multiplicative relation is found, it is added to the rhizome (lines 12 through 16). A recursive call is made to analyze each multiplicative variable in turn (lines 17 through 18).

4.3.2 SCRIPT_DECOMPRESSOR algorithm

Based on character decomposition, algorithm 4.6 creates new Script expressions from input Script expression and adds additive relations between those new Script expressions. Each new Script expression becomes an input to algorithm 4.5 for further analysis.

Algorithm 4.6 Algorithm SCRIPT_DECOMPRESSOR

```
input  : ($s$) where $s$ is a script expression
output: Updated global variable $G = (V, E)$
88 $A$ ;                                  // data structure of type Chars
89 $B$ ;                                  // set of script expressions
90 $C$ ;                                  // set of strings
91 begin
92     $A \longleftarrow$ Parse($s$, Ⓒ)
       for $0 \leq i < |A|$ do
93         $B \longleftarrow \emptyset$
           $C \longleftarrow$ Expand($A[i]$)
           for $0 \leq j < |C|$ do
94             $\bar{A} \longleftarrow A$
               $\bar{A}[i] \longleftarrow C[j]$
               $B \overset{+}{\longleftarrow}$ SCRIPT_GENERATOR(CreateAlgebraic($\bar{A}$))
95         if $|B| > 0$ then
96             if $B \cap$ RestrictedScript $= \emptyset$ then
97                 $G(V, E) \overset{+}{\longleftarrow}$ AdditionRelations($s, B$)
98             for $b \in B$ do
99                 RHIZOME($b$)
```

The input Script is first analyzed to determine all layer 0 characters, which are stored in `Chars` data structure (line 5). Each character is then analyzed in turn (lines 6 through 17). Each character is expanded (line 8) and each expanded character replaces the character from which it was obtained (lines 10 through 11). A valid Script expression is regenerated from this modified Script expression (which may be invalid) and stored in a temporary variable B (line 12). If B does not contain any members, no further analysis takes place and the algorithm terminates. Otherwise, additive relations between modified Scripts are added to the rhizome (line 15) and a call to algorithm 4.5 is made (line 17).

4.3.3 Key construction

Algorithms 4.5 and 4.6 are used with the following settings in order to obtain keys:

1. Setting the `MinLayer` parameter to $L_n - 1$ will limit the analysis to a computation of additive order and symmetry relations for layer L_n as well as multiplicative order relations for layer $L_n - 1$. Lines 3 through 4 of algorithm 4.5 depend on this setting.

2. Setting the `MultSym` parameter to logical $false$ will disable the creation of multiplicative symmetry relations. Lines 14 through 16 of algorithm 4.5 depend on this setting.

3. Setting the `MultOrd` parameter to one of the allowed values (see section 4.1.3, in particular the `MultiplicationOrder` function) will control which multiplication order relations are created.

4. The parameter `RestrictedScript` defines a list of restricted scripts. Any script that belongs to this list should not be analyzed. Lines 5, 12, 20 of algorithm 4.5 and line 14 of algorithm 4.6 depend on this setting.

4.4 From the Rhizome to the Semantic Sphere

The typology of words, clauses, phrases and non-grammatical constructions have been explained in 3.2.8. Given an IEML Script expression s, the algorithm 4.7 is used to generate all clauses and phrases. Algorithm 4.7 delegates all computation to algorithms 4.5, 4.8, 4.9 and 4.10. The complexity of algorithm 4.7 is therefore proportional to the complexity of the underlying algorithms. Algorithm 4.7 uses algorithm 4.5 with the following setting:

- The parameter `Terms` defines a list of Script expressions. Any Script that belongs to that list *must not* be analyzed. Lines 5 through 6 of algorithm 4.5 depend on this setting.

Algorithm 4.7 Algorithm ENUNCIATION

```
input  : ($s$) where $s$ is a Script expression
output: Global variables $G = (V, E)$, $W'$, $W''$, $C^1$, $C^{k+1}$ and $P^k$
begin
    $G$ = RHIZOME($s$) ;                          // computes $G = (V, E)$
    WORDS($G$) ;                                  // computes $W'$ and $W''$
    CLAUSE($G$) ;                                 // computes $C^1$
    PHRASE($G$) ;                                 // computes $C^{k+1}$ and $P^k$
```

The following sections describe algorithms 4.8, 4.9 and 4.10. For a description of algorithm 4.5, please refer to section 4.3.

4.4.1 WORDS algorithm

Algorithm 4.8 traverses the rhizome in order to detect *words*. There are two types of words, stored in W' and W'', which differ in the type of relations they

`clause`
$SEE \otimes EEE \otimes SES$
$SES \otimes EEE \otimes SEE$
$SES \otimes EEE \otimes SES$
$SES \otimes EEE \otimes EEE$
$S \otimes S \otimes SES$
$S \otimes SES \otimes S$
$S \otimes SES \otimes SES$
$SES \otimes S \otimes S$
$SES \otimes S \otimes SES$
$SES \otimes SES \otimes S$
$SES \otimes SES \otimes SES$

Table 4.3: Templates defined by variable `clause`

belong to and in the expression pattern they exhibit. The rhizome traversal itself is implemented in algorithm 4.11 (lines 2 and 4). The results of the traversal are then transformed by a global function on lines 3 and 5. The complexity of this algorithm is directly proportional to the complexity of algorithm 4.11.

Algorithm 4.8 Algorithm WORDS

```
input  : $(G)$ where $G$ is a graph $G = (V, E^A \cup E^B \cup E^C \cup E^D)$
output : Global variables $W'$ and $W''$
begin
    TRAVERSE($G$, $V$, $E^A$, $E^B$, $W'$, Terms) ;                  // traverse graph
    $W' \longleftarrow \{w \in W' \mid$ DetType($w$)$\}$ ;            // transform result
    TRAVERSE($G$, Pattern(‘SES‘), $E^C$, $E^D$, $W''$, Terms)
    $W'' \longleftarrow \{w \in W'' \mid$ DetType($w$)$\}$ ;          // transform result
```

4.4.2 CLAUSE algorithm

The purpose of algorithm 4.9 is to find all clauses of layer 1. This is accomplished by considering different starting points for rhizome traversal (either members of `Terms` or of $W' \cup W''$) and by searching for scripts that conform to a particular pattern (by supplying different templates for the function `Pattern`) on lines 2 and 3. The complexity of this algorithm is directly proportional to the complexity of algorithm 4.11.

4.4.3 PHRASE algorithm

The purpose of algorithm 4.10 is to find all clauses of layer 2 and above, and all phrases of layer 1 and above.

Algorithm 4.10 computes all the phrases (line 3), while line 5 computes all the clauses of layer 2 and above. The logic of this algorithm logic is very similar

Algorithm 4.9 Algorithm CLAUSE

```
input  : $(G)$ where $G$ is a graph $G = (V, E^A \cup E^B \cup E^C \cup E^D)$
output : Global variable $C^1$
109 begin
110     TRAVERSE($G$, Pattern(‘SES‘ | ‘SEE‘), $E^C$, $E^D$, $C^1$, Terms)
        TRAVERSE($G$, Pattern(‘clause‘), $E^C$, $E^D$, $C^1$, $W' \cup W''$)
```

to the logic of algorithm 4.11. Its complexity in time and space is a scalar multiple (proportional to the number of layers) of the complexity of algorithm 4.11.

Algorithm 4.10 Algorithm PHRASE

```
input  : $(G)$ where $G$ is a graph $G = (V, E^A \cup E^B \cup E^C \cup E^D)$
output : Global variables $C^{k+1}$ and $P^k$
111 begin
112     for $1 \leq i \leq 6$ do
113         TRAVERSE($G$, $V$, $E^A$, $E^B$, $P^i$, $C^i$)
            if $i < 6$ then
114             TRAVERSE($G$, Pattern(‘SSS‘ | ‘SES‘ | ‘SEE‘), $E^C$, $E^D$, $C^{i+1}$, $P^i$)
```

4.4.4 TRAVERSE algorithm

Algorithm 4.11 is a generic helper algorithm that is responsible for traversing a graph and grouping those vertices that conform to certain criteria. This operation occurs frequently in the transcoding process, and thus it is helpful to abstract it out from the rest of the process in order to make algorithms 4.8 , 4.9 and 4.10 more readable.

Algorithm 4.11 runs for all members w of set A_w (line 2). Local variables V_r and $V^w_{A_q}$ are initialized respectively with all vertices of the graph whose relations with w are contained in A_r, and all vertices of the graph that are not present in A_s and whose relations with w are contained in A_q (lines 3 and 4). The algorithm then iterates over all $V^w_{A_q}$ members (line 5). A local variable V_q stores all vertices of the graph whose relations with $v^w_{A_q}$ are contained in A_q (line 6). As a final step, the algorithm stores a variable $v^w_{A_q}$ in the set A_s if the set $V_q \cap V_r$ is a subset of set A_w and the variable $v^w_{A_q}$ belongs to the set A_p (lines 7 through 9).

Since it is a generic algorithm, the following constraints must be met:

1. Members of V_q, V_r, A_w, A_p and A_s are of the same type as $v^w_{A_q}$
2. Members of A_r are sets

Algorithm 4.11 Algorithm TRAVERSE

input : $(G, A_p, A_q, A_r, A_s, A_w)$ where G is a graph $G = (V, E)$ and A_p, A_q, A_r, A_s, A_w are sets

output: Updated variable pointed to by A_s

```
begin
    for $\forall w \in A_w$ do
        $V_r \longleftarrow \{v \in V \mid \{w, v\} \in A_r\}$ ;                          // subset of $V$
        $V^w_{A_q} \longleftarrow \{v \in V \mid v \notin A_s \ \wedge \ (v, w) \in A_q\}$ ;   // subset of $V$
        for $\forall v^w_{A_q} \in V^w_{A_q}$ do
            $V_q \longleftarrow \{v \in V \mid (v^w_{A_q}, v) \in A_q\}$ ;                // subset of $V$
            if $V_q \cap V_r \subseteq A_w$ then
                if $v^w_{A_q} \in A_p$ then
                    $A_s \overset{+}{\longleftarrow} v^w_{A_q}$
```

3. Members of A_q are ordered pairs.

Here we are operating within the context of the transcoding process. This means that members of the variables in the first constraint are vertices. Members of the variable in the second constraint are relations of additive symmetry or multiplication (edges of the graph). Finally, the members of the variable in the third constraint are relations of additive or multiplicative order, which are represented by directed edges as they denote parent-child relations.

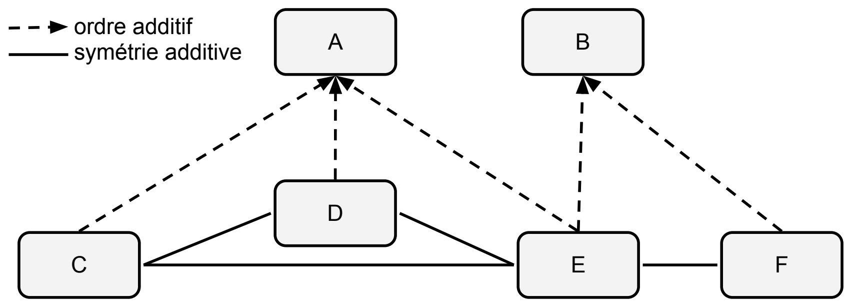


Figure 4.1: Additive order and symmetry relations

As an example, consider vertex E in figure 4.1. Given a graph

$$G = (A_w, A_r \cup A_q) \tag{4.2}$$

where

$$A_w = \{A, B, C, D, E, F\} \tag{4.3}$$

$$A_r = \{\{C, D\}, \{C, E\}, \{D, E\}, \{E, F\}\} \tag{4.4}$$

$$A_q = \{(A, C), (A, D), (A, E), (B, E), (B, F)\} \tag{4.5}$$

and $A_s = \emptyset$, $A_p = A_w$.

If we run algorithm 4.11 it will yield the following: $V_r = C, D, F$ (line 3), $V^w_{A_q} = A, B$ (line 4). We now have two cases to consider: in the first case, $v^w_{A_q} = A$, $V_q = C, D, E$, conditions on lines 7 and 8 are met, and we store A in A_s; in the second case, $v^w_{A_q} = B$, $V_q = E, F$, conditions on lines 7 and 8 are met, and we store B in A_s. However, if F were not contained in A_w, A but not B would be stored in A_s.

Assuming that sets are implemented as hash tables, the time complexity of Algorithm 4.11 is:

$$T_1 = |V^w_{A_q}| * ((min(|A_q|, |V|) + min(|V_q|, |V_r|, |A_w|)) \tag{4.6}$$

for lines 5 through 9, and:

$$T_2 = |A_w| * (min(|A_r|, |V|) + min(|A_q|, |V|) + T_1) \tag{4.7}$$

overall, where $|...|$ denotes the size of a particular set. An upper limit for this equation is given by $|V|^3$ as V becomes large. The space complexity is then proportional to $|V|$.

4.4.5 SEMANTIC_NETWORK algorithm

Algorithm 4.12 creates data structures that are needed to represent the semantic network in a visual form. These data structures contain the following elements:

- a mapping from each term to a selected natural language using a specified dictionary (line 3),

- three graphs defining, respectively, the relations between terms (line 5), relations between graphs (line 9 and 14) and, finally, a possibly disjoint graph (line 17).

These data structures define the necessary and sufficient information to represent a semantic network.

Line 5 adds vertices and edges representing a complete graph between all sequences in additive symmetry relations. Line 9 adds vertices and edges, and line 14 adds vertices and labelled edges, where vertices are the substance and attribute semes of a sequence and the label of the edge is the mode seme of the sequence (if the mode seme is empty, then it can be omitted). Finally, line 17 groups specific graphs together.

Algorithm 4.12 Algorithm SEMANTIC_NETWORK

```
input  : ($T$) where $T$ is the natural language mapping
output: Global variables $G_{SEM}(V,E)$, $\bar{G}_{SEM}(V,E)$, $\bar{\bar{G}}_{SEM}(V,E)$
begin
    for $\forall t \in$ Terms do                                        // all terms
        $H_{NAT} \xleftarrow{+} \{t, \texttt{Translate}(t,T)\}$ ;       // term to language map
    for $\forall w \in W'$ do
        $G_{SEM} \xleftarrow{+} \texttt{Clique}(w)$ ;                    // w's complete graph
    for $\forall w \in W''$ do
        $v \longleftarrow \{w_{sub}, w_{mod}\}$ ;                        // vertice
        $e \longleftarrow \{(w_{sub}, w_{mod})\}$ ;                      // edge
        $\bar{G}_{SEM}(V,E) \xleftarrow{+} (v,e)$
    for $\forall c \in \bigcup_{i=0}^{6} C^i$ do
        $v \longleftarrow \{w_{sub}, w_{att}\}$ ;                        // vertice
        $e \longleftarrow \{(w_{sub}, w_{att})\}$ ;                      // edge
        $e \xleftarrow{+} (w_{mod})$ ;                                   // edge's label
        $\bar{G}_{SEM}(V,E) \xleftarrow{+} (v,e)$
    for $\forall p \in \{P^i \mid 1 \leq i \leq 6\}$ do
        for $\forall c \in p$ do
            $\bar{\bar{G}}_{SEM}(p) \xleftarrow{+} \bar{G}(c)$
```

Chapter 5

Mathematical formalization

«Let no one untrained in geometry enter.» Plato

5.1 Introduction

The model of the IEML language and of its semantic variables is presented in section 5.2. I show that IEML is a *regular language*, a class of languages that is extremely efficient at computational tasks involving sequencing and repetition, and is furthermore recognized by *finite state machines.*

The *group structure* describes the mathematical concept of symmetry, where symmetry can be understood as invariance under some transformations. Symmetry allows us to recognize similarities, and also to discover which properties of elements do not change under transformations. In section 5.3 I show that that IEML semantic variables possess characteristics of *groups* and of *rings.*

In section 5.4 I consider the *computability* of transformations applied to the IEML language. Using finite state machines as the underlying computational model, I show that transformations defined for the semantic variables of the IEML language are computable.

While IEML language is used to create IEML expressions, semantic graphs are a representation of the *semantic relations* that occur within IEML expressions. The model of IEML relations is presented in section 5.5. The combination of a plurality of semantic relations results in *semantic circuits.* In section 5.6 I show that semantic circuits form a groupoid, and I present algorithms for functions that are applicable to rhizomes, a type of semantic circuits.

Section5.7 introduces quantitative criteria used to compare any two semantic circuits. These criteria can be used as a basis for defining a notion of *semantic distance* between two semantic circuits.

I wish to thank Andrew Roczniak, PhD, who worked with me to formalize IEML over the course of ten years, and Nick Soveiko, PhD, who reviewed the final version of this mathematical chapter and proposed the use of the spectral

graph theory to compute similarities between semantic circuits. Nevertheless, I remain solely responsible for any errors or inconsistencies.

5.2 IEML language

5.2.1 A model of IEML Language

Let Σ be a non-empty and finite set of symbols, $\Sigma = \{S, B, T, U, A, E\}$. Let a string s be a finite sequence of symbols chosen from Σ. The length of this string is denoted by $|s|$. An empty string ϵ is a a string with zero occurrence of symbols and its length is $|\epsilon| = 0$. The set of all strings of length k composed with symbols fromΣ is defined as $\Sigma^k \triangleq \{s : |s| = k\}$. The set of all strings above Σ is defined as:

$$\Sigma^* \triangleq \Sigma^0 \cup \Sigma^1 \dots \tag{5.1}$$

The IEML language above Σ is a subset of Σ^*, $L_{IEML} \subseteq \Sigma^*$ where $L = 6$:

$$L_{IEML} \triangleq \left\{s \in \Sigma^* \mid 0 \leq l \leq L,\ |s| = 3^l\right\} \tag{5.2}$$

Proposition 5.2.1. *IEML language as given in equation 5.2 is a regular language [KLE 1956].*

Proof. Consider the definition of regular languages:

- $L = \{\emptyset\}$ and $L = \{\epsilon\}$ are regular languages,
- $L = \{\sigma \mid \sigma \in \Sigma\}$ are regular languages,
- if L_1 and L_2 are regular languages, then so are $L_1 \cup L_2$ and $L_1 \cdot L_2$ (concatenation).

Since L_{IEML} can be constructed from its alphabet $\Sigma = \{S, B, T, U, A, E\}$ and using only statements from the above definition, it is a regular language. □

We note that any string belonging to the L_{IEML} language can be also obtained from the Σ alphabet by application of the *triplication* function. String concatenation takes two strings and produces a third string, which is composed of symbols of the first string followed by the symbols of the second string. The IEML triplication function is a specialization of the string concatenation, where three strings (a, b, c) belonging to L_{IEML} and of the same length are concatenated and where the length of each string is at most 3^{L-1}:

$$f_t(a, b, c) \triangleq (abc\ :\ |a| \leq 3^{L-1} \wedge\ |a| = |b| = |c|\ \wedge\ a, b, c \in L_{IEML}) \tag{5.3}$$

Regular languages are extremely efficient at computational tasks involving sequencing and repetition and can be recognized by finite state machines, which are discussed in section 5.4.1.1.

5.2.2 Model of Semantic Sequences

Definition 5.2.1. *A string s is a semantic sequence if and only if $s \in L_{IEML}$.*

Unless otherwise specified, "sequence" and "semantic sequence" are used interchangeably in the remainder of the text. To denote the p_n'th primitive of a sequence s, we use a superscript n (where $1 \leq n \leq 3^l$) and we write s^n. Note that for any sequence s of layer l, s^n is undefined for any $n > 3^l$. Two semantic sequences are distinct if and only if either of the following holds: a) their layers are different; b) they are composed from different primitives; c) their primitives do not follow the same order: for any s_a and s_b,

$$s_a = s_b \iff \forall n,\ s_a^n = s_b^n \ \wedge\ |s_a| = |s_b| \tag{5.4}$$

Let us now consider binary relations between semantic sequences in general. These are obtained by performing a Cartesian product of two sets[1]. For any set of semantic sequences X, Y (where $s_a \in X$, $s_b \in Y$ and equation 5.4 is used), we define four binary relations (*whole* $\subseteq X \times Y$, *substance* $\subseteq X \times Y$, *attribute* $\subseteq X \times Y$ and *mode* $\subseteq X \times Y$) as follows:

$$\texttt{whole} \triangleq \{(s_a, s_b) \mid s_a = s_b\} \tag{5.5}$$

$$\texttt{substance} \triangleq \{(s_a, s_b) \mid 1 \leq n \leq |s_b|,\ s_a^n = s_b^n \ \wedge\ |s_a| = 3|s_b|\} \tag{5.6}$$

$$\texttt{attribute} \triangleq \{(s_a, s_b) \mid 1 \leq n \leq |s_b|,\ s_a^{n+|s_b|} = s_b^n \ \wedge\ |s_a| = 3|s_b|\} \tag{5.7}$$

$$\texttt{mode} \triangleq \{(s_a, s_b) \mid 1 \leq n \leq |s_b|,\ s_a^{n+2|s_b|} = s_b^n \ \wedge\ |s_a| = 3|s_b|\} \tag{5.8}$$

Equation 5.5 states that any two semantic sequences that are equal are in a `whole` relationship (we can also write s_b `whole` s_a). Equations 5.6, 5.7 and 5.8 state that any two semantic sequences that share specific sub-sequences may be in `substance`, `attribute` or `mode` relationship. For any two semantic sequences s_a and s_b, if they are in one of the above relationships, then we say that s_b plays a *role with respect to* s_a and we call s_b a *seme* of sequence:

Definition 5.2.2. *For any semantic sequence s_a and s_b, if*

$$(s_a, s_b) \in \mathit{whole} \cup \mathit{substance} \cup \mathit{attribute} \cup \mathit{mode}$$

then s_b plays a role with respect to s_a and s_b is called a seme.

We can now group distinct semantic sequences together into sets. A useful grouping is based on the layering of the semantic sequences, as discussed in section 5.2.3.

[1]A Cartesian product of two sets X and Y is written as follows: $X \times Y = \{(x, y) \mid x \in X, y \in Y\}$

5.2.3 Model of Semantic Categories

A *category* of L_{IEML} of layer l is a subset of L_{IEML} such that all semantic sequences of that subset have the same length:

$$c_l \triangleq \{s \mid s \in L_{IEML} \ \wedge \ |s| = 3^l\} \tag{5.9}$$

Definition 5.2.3. *A semantic category c is a set containing semantic sequences at the same layer.*

Unless otherwise specified, «*category*» and «semantic *category*» are used interchangeably in the remainder of the text. The layer of any *category c* is exactly the same as the layer of the semantic sequences included in that *category.* The set of all *categories* of layer L is given as the powerset [2] of the set of all strings of layer L of L_{IEML}:

$$C_L \triangleq \mathcal{P}(\{c_L\}) \tag{5.10}$$

Two *categories* are distinct if and only if they differ by at least one element. For any c_a and c_b:

$$c_a = c_b \iff c_a \subseteq c_b \wedge c_b \subseteq c_a \tag{5.11}$$

A weaker condition can be applied to *categories* of distinct layers (since two *categories* are different if their layers are different) and is written as:

$$\ell(c_a) \neq \ell(c_b) \Longrightarrow c_a \neq c_b \tag{5.12}$$

where $\ell(\cdot)$ denotes the layer of a *category.*

Analogously to sequences, we consider binary relations between any *categories* c_i and c_j (where $\ell(c_i), \ell(c_j) \geq 1$). For any set of *categories* X, Y (where $c_a \in X$, $c_b \in Y$ and equations 5.11, 5.6, 5.7 and 5.8 are used), we define four binary relations ($\mathit{whole}_C \subseteq X \times Y$, $\mathit{substance}_C \subseteq X \times Y$, $\mathit{attribute}_C \subseteq X \times Y$ and $\mathit{mode}_C \subseteq X \times Y$) as follows:

$$\texttt{whole}_C \triangleq \{(c_a, c_b) \mid c_a = c_b\} \tag{5.13}$$

$$\texttt{substance}_C \triangleq \{(c_a, c_b) \mid \forall\ s_a \in c_a, \exists s_b \in c_b, (s_a, s_b) \in \texttt{substance}\} \tag{5.14}$$

$$\texttt{attribute}_C \triangleq \{(c_a, c_b) \mid \forall\ s_a \in c_a, \exists s_b \in c_b, (s_a, s_b) \in \texttt{attribute}\} \tag{5.15}$$

$$\texttt{mode}_C \triangleq \{(c_a, c_b) \mid \forall\ s_a \in c_a, \exists s_b \in c_b, (s_a, s_b) \in \texttt{mode}\} \tag{5.16}$$

For any two *categories* c_a, c_b, if they are in one of the above relations, then we say that c_b plays a *role* with respect to c_a and c_b is called a seme of *category:*

[2] A powerset of S is the set of all subsets of S, including the empty set $\emptyset$

Definition 5.2.4. *For any category c_a and c_b, if*

$$(c_a, c_b) \in \mathtt{whole}_C \cup \mathtt{substance}_C \cup \mathtt{attribute}_C \cup \mathtt{mode}_C$$

then c_b plays a role with respect to c_a and c_b is called a seme.

5.2.4 Language of categories

Categories will usually be generated using the following approach. Taking the powerset of Σ (the set of all subsets of Σ, including the empty set $\emptyset$), we represent it by: $\Sigma_{IEML} \triangleq \mathcal{P}(\Sigma)$. Contained in Σ_{IEML} are sets such as $\{S\}$, $\{A\}$, $\{U, A\}$, $\{E\}$, $\{S, B, T\}$, and$\{\emptyset\}$. Note that the order of the members of the set Σ_{IEML} is irrelevant; for example symbols $\{U, A\}$ and $\{A, U\}$ are considered to be one and the same. The set Σ_{IEML} is the set of all *IEML semantic characters*, and we define the *category language* as the language:

$$\bar{\Sigma} \triangleq \Sigma_{IEML} \backslash \{\emptyset\} \tag{5.17}$$

Let a string s be a finite sequence of symbols chosen from $\bar{\Sigma}$. The length of this string is denoted by $|s|$. An empty string ϵ is a string with zero occurrence of symbols, and its length is $|\epsilon| = 0$. The set of all strings of length k composed of symbols from $\bar{\Sigma}$ is defined as $\bar{\Sigma}^k \triangleq \{s : |s| = k\}$. The set of all strings above $\bar{\Sigma}$ is defined as:

$$\Sigma^*_{cat} \triangleq \bar{\Sigma}^0 \cup \bar{\Sigma}^1 \ldots \tag{5.18}$$

The category language above $\bar{\Sigma}$ is a subset of Σ^*_{cat}, $L_{cat} \subseteq \Sigma^*_{cat}$, where $L = 6$:

$$L_{cat} \triangleq \left\{s \in \Sigma^*_{cat} \mid 0 \leq l \leq L,\ |s| = 3^l\right\} \tag{5.19}$$

IEML sequences are obtained from any sequence $s \in L_{cat}$ by performing a Cartesian product between all the symbols of that sequence s. For instance, the sequence $\{S, B, T\}\{U, A\}\{E\}$ gives the IEML set of sequences $\bar{s}$:

$$\bar{s} = \{SUE, SAE, BUE, BAE, TUE, TAE\} \tag{5.20}$$

5.2.5 Semantic category types

Let us now introduce certain *types* of categories :

Definition 5.2.5. *A* category *c of layer l is* singular *under the following conditions: $|c| = 1$ and the sequence $s \in c$ is composed of symbols s^n, where $s^n \in \bar{\Sigma} \ \wedge\ |s^n| = 1$ for $1 \leq n \leq 3^l$.*

All *categories* that are not singular are *plural.*

Definition 5.2.6. *A category c of layer l is simple under the following conditions : $|c| = 1$ and the sequence $s \in c$ is composed of symbols s^n, where $s^n \in \bar{\Sigma}$ for $1 \leq n \leq 3^l$.*

All categories that are not simple are *complex*. Note that all the singular categories are also simple, because $\bar{\Sigma}$ contains all the symbols that have only a single member ($\{S\}$, $\{B\}$, $\{T\}$, $\{U\}$, $\{A\}$, $\{E\}$).

5.2.6 Model of Catsets and USLs

A catset is a set of distinct *categories* of the same layer:

Definition 5.2.7. *A catset κ is a set containing categories $\kappa_l = \{c \mid l(c) = l\}$ such that $\forall a, b \in \kappa_l,\ a \neq b$*

The layer of a catset[3] is given by the layer of any of its members: if some $c \in \kappa$, then $\ell(\kappa) = \ell(c)$. A USL is composed of up to seven catsets of different layers:

Definition 5.2.8. *A USL u is a set containing catsets of different layers: $u = \{\kappa\}$ such that $\forall a, b \in u,\ l(a) \neq l(b)$*

Note that since there are 7 distinct layers, a USL can have at most seven members. All standard set operations on USLs are always performed on sets of *categories* (and therefore on sets of sequences), layer by layer.

5.3 Symmetry Properties

5.3.1 Generalities

The mathematical concept of *group* consists of a *set,* a *binary operation*, and some *properties* when the operation is applied to members of the set. The binary operation $\circledast$ on a given set S associates, to elements x and y of S, a third element: $x \circledast y$ of S. The properties are associativity ($\forall x, y, z \in S,\ (x \circledast y) \circledast z = x \circledast (y \circledast z)$), the existence of an identity element, and the existence of an inverse element for each element of S.

By denoting the identity element of a group S as x^1 and the inverse element as x^{-1} for each $x \in S$, we obtain the following basic properties of groups:

Proposition 5.3.1. *A group $(S, \circledast)$ has exactly one identity element x^1 that satisfies $\forall x \in S,\ x \circledast x^1 = x^1 \circledast x = x$*

Proof. Assuming that a given $i \in S$ has the property $i \circledast x = x,\ \forall x \in S$, then we have $i = i \circledast x^1 = x^1$. Similarly, if $i \in S$ has the property $x \circledast i = x,\ \forall x \in S$, then we have $i = x^1 \circledast i = x^1$. $\square$

Proposition 5.3.2. *A group $(S, \circledast)$ has exactly one inverse element x^{-1} for each $x \in S$.*

[3]Note that a *category* c can be written as $c \in C_L$, while a catset κ can be written as $\kappa \subseteq C_L$

Proof. Among the group properties, there exists an element $x^{-1} \in S$ with the property $x \circledast x^{-1} = x^{-1} \circledast x = x^1$, $\forall x \in S$. For any $i \in S$ with the property $x \circledast i = x^1$, then $i = x^1 \circledast i = (x^{-1} \circledast x) \circledast i = x^{-1} \circledast (x \circledast i) = x^{-1} \circledast x^1 = x^{-1}$. Similarly, for any element with the property $i \circledast x = x^1$, then $i = x^{-1}$, which implies that the inverse of any $x \in S$ is uniquely determined. □

Proposition 5.3.3. *For any elements x, y of a group $(S, \circledast)$, $(x \circledast y)^{-1} = y^{-1} \circledast x^{-1}$*

Proof. We have verified that $(x \circledast y) \circledast (y^{-1} \circledast x^{-1}) = x \circledast (y \circledast (y^{-1} \circledast x^{-1})) = x \circledast (y \circledast y^{-1}(\circledast x^{-1})) = x \circledast (x^1 \circledast x^{-1}) = x \circledast x^{-1} = x^1$. A similar result is obtained for $(y^{-1} \circledast x^{-1}) \circledast (x \circledast y)$, which implies that the inverse of $(x \circledast y)$ is $y^{-1} \circledast x^{-1}$. □

The group structure can be extended by adding operations (which leads to a two-operation structure called a *ring*) or by adding properties. For example, the addition of the commutativity property - $x \circledast y = y \circledast x$, $\forall x, y \in S$ - creates an A*belian* group. The removal of properties leads to simpler structures called *groupoids.*

5.3.2 Semantic Categories

For any *category* c we can define a function $f : c \to c$ that is injective ($\forall x, y \in c, f(x) = f(y) \to x = y$, a one-to-one function) and total ($\forall x \in c,\ f(x) \in c$). We gather all distinct functions in a set:

$$F_c = \{f_i \,|\exists x \in c, i \neq j, f_i(x) \neq f_j(x)\} \tag{5.21}$$

There are $|c|!$ such functions, and they represent all the possible permutations of the set c. For any functions f_i,$f_j \in F_c$, the output of one can be used as input for the other, resulting in function composition: $(f_i \circ f_j)(s) \triangleq f_i(f_j(s))$. Let us now consider *symmetry groups* [BUT 1991].

Proposition 5.3.4. *The group $G = (F_c, \circ)$, where the group operation is the function composition, is a symmetry group.*

Proof. Closure: clearly, successive applications of any $f \in F_c$ form a one-to-one mapping from the set c to the set c. Since F_c contains all one-to-one functions for the set c, then $f_i \circ f_j = f_k \in F_c$.

Associativity: using the definition of function composition, $\forall f_i, f_j, f_k \in F_c, ((f_i \circ f_j) \circ f_k)(s) = (f_i(f_j(f_k(s))))$. On the other hand, $(f_i \circ (f_j \circ f_k))(s) = (f_i(f_j(f_k(s))))$.

Identity: let $f_0(s) = s, \forall s \in c$. Clearly $f_0 \in F_c$. We now have $(f_0 \circ f_i)(s) = f_0(f_i(s)) = f_i(s)$, and on the other hand $(f_i \circ f_0)(s) = f_i(f_0(s)) = f_i(s)$.

Inverse: let $f_i(s) = z, \forall f_i \in F_c$ We define $f_i^{-1}(z) = s$. Accordingly, $(f_i \circ f_i^{-1})(z) = f_i(f_i^{-1}(z)) = z = f_0(z)$, and on the other hand, $(f_i^{-1} \circ f_i)(s) = f_i^{-1}(f_i(s)) = s = f_0(s)$. Since $f_i^{-1}(z)$ is a one-to-one function on c, then $f_i^{-1}(z) \in F_c$. □

5.3.3 Catsets and USLs

From definition 5.2.7, a catset contains distinct *categories* of the same layer. Since the language L_{IEML} is finite, the number of distinct *categories* of the same layer is given by the size of the set C_L. In general then, by considering C_L as a group, we obtain a special type called a *ring,* [BUT 1991] as shown in the following proposition:

Proposition 5.3.5. *The group $G = (C_L, \oplus, \otimes)$, with the set difference (Δ) operation and the set intersection $(\cap)$ operation, is a ring.*

Proof. Symmetric set difference results in members that are in either set, but not in both: for sets A and B, the difference is $(A \backslash B) \cup (B \backslash A)$. Then $\forall A, B, C \subseteq C_L$:

The group $(C_L, \oplus)$ is an *Abelian group* since $\forall c_i, c_j \in C_L,\ c_i \oplus c_j \in C_L$ by the powerset definition (closure), $(A \oplus B) \oplus C = A \oplus (B \oplus C)$ by associativity of the set union operation (associativity), $A \oplus B = B \oplus A$ by commutativity of the set union operation (commutativity), $\exists c_0 \in C_L,\ \forall c_i \in C_L,\ c_i \oplus c_0 = c_0 \oplus c_i = c_i$ where $c_0 = \emptyset$ (identity), and $\exists c_i^{-1} \in C_L,\ \forall c_i \in C_L,\ c_i \oplus c_i^{-1} = c_i^{-1} \oplus c_i = c_0$ where $c_i^{-1} = c_i$ (inverse) .

The group $(C_L, \otimes)$ is a *monoid* since $\forall c_i, c_j \in C_L,\ c_i \otimes c_j \in C_L$ by the powerset definition (closure), $(A \otimes B) \otimes C = A \otimes (B \otimes C)$ by associativity of the set intersection operation (associativity), and $\exists c_0 \in C_L,\ \forall c_i \in C_L,\ c_i \otimes c_0 = c_0 \otimes c_i = c_i$ where $c_0 = \emptyset$ (identity).

The multiplication distributes over the addition operation: $\forall c_i, c_j, c_k \in C_L$, $c_i \otimes (c_j \oplus c_k) = (c_i \otimes c_j) \oplus (c_i \otimes c_k)$ and $(c_i \oplus c_j) \otimes c_k = (c_i \otimes c_k) \oplus (c_j \otimes c_k)$ by distributivity of the set intersection operation over the set symmetric difference operation. □

From definition 5.2.8, a USL is a set of catsets of different layers. Since at each layer L there are $|C_L|$ distinct catsets, the whole semantic space is defined by the tuple

$$\mathcal{USL} = C_0 \times C_1 \times C_2 \times C_3 \times C_4 \times C_5 \times C_6 \tag{5.22}$$

Considering $\mathcal{USL}$ as a group, we obtain the following proposition:

Proposition 5.3.6. *The group $G_{\mathcal{USL}} = (\mathcal{USL}, \oplus, \otimes)$, with the symmetric set difference (Δ) operation and the set intersection $(\cap)$ operation applied at each layer, is a ring [BUT 1991].*

Proof. The proof follows the proof for proposition 5.3.5. □

5.3.4 Transformational Symmetry

We describe transformational symmetries using the concept of *categories* [ARZ 1999],

$$\mathbb{C} = (O, M, \circ) \tag{5.23}$$

where O is a collection of objects, M a collection of morphisms between two members of the collection of objects, and $\circ$ a binary composition operation between compatible morphisms. The Category C has the following properties : $\forall a, b, c \in O$, if $u = (a \to b) \in M$ and $v = (b \to c) \in M$ then $\exists w = u \circ v = (a \to c) \in M$; $\forall u, v, w \in M$, and if $(u \circ v) \circ w \in M$ then $(u \circ v) \circ w = u \circ (v \circ w) \in M$; $\forall a \in O$, $\exists i_a \in M$ such that $\forall u = (a \to b)$, $i_a \circ u = u \ \wedge \ u \circ i_a = u$.

5.3.4.1 IEML language as a category

If we consider objects in the collection O to be subsets of L_{IEML} from equation 5.2 as an example, $O = \{o_i \mid o_i \subset L_{IEML}; \forall s \in o_i,\ |s| = 3^i;\ 0 \leq i \leq 6\}$ then morphisms are given by $M = \{m_i \mid m_i = (o_i \to o_{i+1});\ 0 \leq i \leq 5\}$. The specific function represented by the morphism is the *triplication*[4] total function, $f_t : o_i \to o_{i+1}$ defined in equation 5.3.

5.3.4.2 Role permutation symmetry

The collection M in equation 5.23 also contains an automorphism $m_r = (o_i \to o_i)$, represented by a unary operation acting on specified roles for each sequence of a semantic category (see section 5.2.3).

5.3.4.3 Seme permutation symmetry

The collection M in equation 5.23 also contains an automorphism $m_s = (o_i \to o_i)$, represented by a binary operation acting on the specified role and specified semes for each sequence of a semantic category (see section 5.2.3).

5.4 Computability

5.4.1 Generalities

In order to demonstrate the computability of semantic transformations $f : S \to S$, where S represents semantic variables (either categories, catsets or USLs), we will use the finite state machine formalism. All operations based on transformations of semantic variables are computable by following the properties of finite state machines.

5.4.1.1 Finite state machines

A finite state machine (FSM) can be defined [GIL 1962] as follows:

Definition 5.4.1. *(FSM) A finite state machine M is a synchronous system represented by a quintuple $M = (\Sigma, \Gamma, Q, \delta, \omega)$ with a finite input alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_i\}$, a finite output alphabet $\Gamma = \{\gamma_1, \ldots, \gamma_j\}$, a finite state set $Q = \{q_1, \ldots, q_n\}$ and a pair of characterizing functions δ and ω given by $q_{v+1} = \delta(q_v, \sigma_v)$, $\gamma_v = \omega(q_v, \sigma_v)$ where σ_v, γ_v, q_v are the input symbol, output symbol and state of M at $v = 1, 2, \ldots$*

[4]multiplication

Thus, the input to the machine is a sequence of symbols $\sigma_1 \dots \sigma_k$, and the output of the machine is a sequence of symbols $\gamma_1 \dots \gamma_k$.

In general, machines may change state even if there is no input, they may have multiple starting states, and there may be zero, one or more executable transition rules. To accommodate these differences a transition mapping is defined as $\delta_n : Q \times \Sigma^* \to \mathcal{P}(Q)$, where the return value is represented as a powerset of Q. Similarly, the output mapping can be defined as $\omega_n : Q \times \Sigma^* \times Q \to \Gamma^*$. In those non-deterministic cases, an alternative notation is more convenient, and instead of the mappings δ_n and ω_n, the transition relation Δ is used:

Definition 5.4.2. *(NFSM) A non-deterministic finite state machine M is a tuple $M = (\Sigma, \Gamma, Q, Q_0, \Delta, F)$ where Σ is a finite input alphabet, Γ is a finite output alphabet, Q is a finite set of states, $Q_0 \subseteq Q$ are the initial states of the machine, $\Delta \subset Q \times \Sigma^* \times \Gamma^* \times Q$ is the transition relation and F represents the acceptance condition where $F \subseteq Q$.*

Each transition relation $(p, \sigma, \gamma, q) \in \Delta$, or edge from state p to state q can be represented as $p \xrightarrow{\sigma|\gamma} q$, where σ and γ play the role of input and output labels of the edge, respectively. Each input to the machine given an initial state will describe a *path* which is a finite sequence [BEA 2002] of relations represented as: $q_0 \xrightarrow{\sigma_1|\gamma_1} q_1 \xrightarrow{\sigma_2|\gamma_2} q_2 \cdots \xrightarrow{\sigma_n|\gamma_n} q_n$.

The input and output labels of the path are

$$\sigma_{in} = \sigma_1 \sigma_2 \dots \sigma_n \tag{5.24}$$

and

$$\gamma_{out} = \gamma_1 \gamma_2 \dots \gamma_n \tag{5.25}$$

respectively. The machine thus computes the relation $(\sigma_{in}, \gamma_{out})$ on $\Sigma^* \times \Gamma^*$. A path is called *successful* if it starts from an initial state $q_s \in Q_0$ and ends in some state q_f that fulfils an acceptance condition, usually defined as $q_f \in F \subseteq Q$. In general then, and in contrast to deterministic machines, a non-deterministic machine can have multiple successful paths for the same input label σ_{in} resulting in a set of relations

$$R = \{(\sigma_{in}, \gamma_{out}), (\sigma_{in}, \kappa_{out}) \dots (\sigma_{in}, \eta_{out})\} \tag{5.26}$$

The description of a finite state machine is usually given in a transition table (table 5.1) or transition graph (figure 5.1) [HOL 1991] from which the δ and ω functions can be obtained. For example, the same machine can be represented by a transition table where rows are a subset of $(Q \times \Sigma \times Q \times \Gamma)$ relations, or a transition graph where edges are labelled with a subset of $(\Sigma \times \Gamma)$ relations.

Condition		Result	
State i	Input	State $i+1$	Output
q_0	-	q_2	1
q_1	-	q_0	0
q_2	0	q_3	0
q_2	1	q_1	0
q_3	0	q_0	0
q_3	1	q_1	0

Table 5.1: Representation of a machine using a transition table

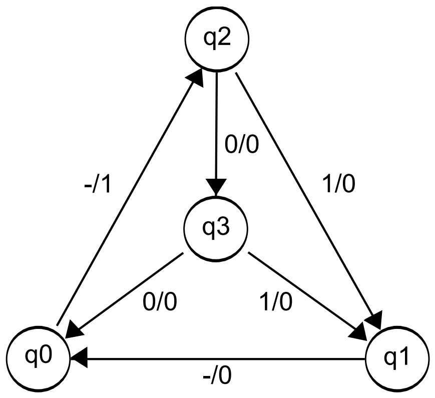


Figure 5.1: Representation of a machine using a graph

5.4.1.2 Finite state automata

A special form of the finite state machine is a deterministic automaton $A_D = (\Sigma, Q, q_0, \delta, F)$. An extended transition function $\hat{\delta}$ that returns a state p when starting from any state q given any valid input sequence w is given by:

$$\hat{\delta}(q, w) = p \tag{5.27}$$

where $q \in Q$ and $w \in \Sigma^*$. This function is defined for all automata, and we show this by induction on the length of the input string:

Basis: $q = \hat{\delta}(q, \epsilon)$, which ensures that there is no state transition when there is no input;

Induction: set w as ua, where a is the last symbol and u is the original string without the last symbol. Then,

$$\hat{\delta}(q, ua) = \delta(\hat{\delta}(q, u), a) \tag{5.28}$$

Operation	*Representation*
Union	$L(A_k) = L(A_i) \cup L(A_j)$
Intersection	$L(A_k) = L(A_i) \cap L(A_j)$
Difference	$L(A_k) = L(A_i) - L(A_j)$
Complementing	$L(A_k) = \Sigma^* - L(A_j)$
Concatenation	$L(A_k) = L(A_i)L(A_j)$
Kleene-star	$L(A_k) = L(A_i)^*$

Table 5.2: Operations that result in regular languages

A language[5] L of A_D denoted $L(A_D)$ [HOP 2001] then is:

$$L(A_D) = \{w \mid \hat{\delta}(q_0, w) \in F\} \tag{5.29}$$

We can define the state equivalence[6] in terms of the set F and equation 5.27 as follows. Two states $p, q \in Q$ are equivalent if and only if $\forall s \in \Sigma^*$, $\hat{\delta}(p, s) \in F \wedge \hat{\delta}(q, s) \in F$. In general, two automata are equivalent if $L(A_i) = L(A_j)$, that is, $L(A_i) \subseteq L(A_j)$ and $L(A_j) \subseteq L(A_i)$. Looking forward, state equivalence gives us a method to determine whether USLs are equivalent.

Languages accepted by finite-state automata are called regular languages. Given two languages $L(A_i)$ and $L(A_j)$, then the operations shown in Table 5.2 result in some other regular language accepted by automaton A_k [HEN 1968]. The Kleene-star of a language represents the set of those strings that can be obtained by taking any number of strings from that language and concatenating them.

5.4.1.3 Finite state transducers

Depending on the problem at hand, finite-state transducers can be interpreted in the following ways [ROC 1997]. We can view finite-state transducers as finite-state automata with an alphabet $\Sigma = \Sigma_1 \times \Sigma_2$, accepting or rejecting string pairs (a, b) where $a \in \Sigma_1^*$ and $b \in \Sigma_2^*$. This automaton is given by $A = (\Sigma, Q, q_0, \delta, F)$ where $F \subseteq Q$ represents some acceptance condition consistent with the problem at hand, and[7]

$$\delta = \{(p, (a, b), q) \mid (p, a, b, q) \in Q \times \Sigma_1^* \times \Sigma_2^* \times Q\} \tag{5.30}$$

We can view finite-state transducers as translators when we consider a class of mappings from strings defined on Σ^* to sets of strings $\mathcal{P}(\Gamma^*)$. This mapping

[5]A *regular expression* can also be used to specify all regular languages. Algorithmic procedures exists to convert from a regular expression to a finite state automaton and vice versa.

[6]If R is a relation on $S \times S$, then R is reflexive if $(a, a) \in R \; \forall \; a \in S$, R is symmetric if $(b, a) \in R$ then $(a, b) \in R$ and R is transitive if $(a, b) \in R$ and $(b, c) \in R$ then $(a, c) \in R$. An equivalence relation ($\equiv$) is reflexive, symmetric and transitive. Equivalence relations give rise to equivalence classes that contain all and only related members.

[7]This can also be seen as a class of directed graphs where states are vertices and labelled edges are given by $E \subseteq Q \times \Sigma_1^* \times \Sigma_2^* \times Q$

is rational if it can be realized by some finite-state transducer. Assuming that $\delta(q,\sigma)$ may return a set of states and using

$$\hat{\omega}(q,s) = \hat{\omega}(q,\sigma s') = \omega(q,\sigma)\hat{\omega}(\delta(q,\sigma),s') = s'' \tag{5.31}$$

we can define a rational transduction $t : \Sigma^* \to \mathcal{P}(\Gamma^*)$ such that

$$t(s_i) = \{s_o \mid \exists\, \hat{\omega}(q_0,s_i), s_i \in \Sigma^*\} \tag{5.32}$$

and a rational function if $|t(s_i)| \leq 1, \forall s_i \in \Sigma^*(|t(s)|$ represents the number of mappings for an input s.)

Finally, a finite-state transducer can be viewed as computing relations between sets of strings[8]. By analogy with the notion that finite-state automata accept languages if and only if they are regular, transducers likewise accept (compute) relations if and only if they are regular.

Regular relations are defined as [KAP 1994]: $\{\emptyset\}$ and $\{(a_i,a_j) \mid a_i,a_j \in A\}$, where A is some automaton and a are strings. Furthermore, if R_a, R_b and R_c are regular relations, then the following also describe regular relations:

1. $R_a \cdot R_b = \{(a_ib_n, a_jb_m) \mid (a_i,a_j) \in R_a \;\wedge\; (b_n,b_m) \in R_b\}$
2. $R_a \cup R_b = \{(r_i,r_j) \mid (r_i,r_j) \in R_a \;\vee\; (r_i,r_j) \in R_b\}$
3. $\cup_{k=0}^{\infty} R_c^k = \emptyset \cup R_c \cup R_c \cdot R_c \cup R_c \cdot R_c \cdot R_c \ldots$

where R^k are k concatenations of R. By definition, thus, regular relations are closed under the concatenation, union and Kleene-star operations. Operations on regular relations that result in other regular relations are presented in Table 5.3. In contrast to the intersection operation on regular languages, regular relations are generally not closed under intersection. To use an oft-cited example, two regular relations[9] $R_1 = (a^nb^*, c^n)$ and $R_2 = (a^*b^n, c^n)$ relate a regular language into another regular language, whereas $R_1 \cap R_2 = (a^nb^n, c^n)$ relates a non-regular language into a regular language.

If we use the output of one machine as the input for another, the machines are then connected in series (or cascade). It is possible to combine the individual machines into one equivalent machine through the composition operation. Using the definitions for the finite-state machines $M_i = (\Sigma, H, Q_i, \delta_i, \omega_i)$ and $M_j = (H, \Gamma, Q_j, \delta_j, \omega_j)$, we obtain $M_k = (\Sigma, \Gamma, Q_i \times Q_j, \delta_k, \omega_k)$ where

$$\begin{aligned} &\delta_k((p,q),\sigma) = \{(p^{'},q^{'}) \mid \delta_i(\sigma,p) = p^{'}, \delta_j(\omega_i(\sigma,p),q) = q^{'}\} \\ &\omega_k((p,q),\sigma) = \{\sigma' \mid \sigma' = \omega_j(\omega_i(\sigma,p),q)\} \end{aligned} \tag{5.33}$$

The input strings $L_I(M) = \{s \in \Sigma^* \mid \exists(s,z) \in R(M)\}$ and output strings $L_O(M) = \{z \in \Gamma^* \mid \exists(s,z) \in R(M)\}$ (and thus the input and output automata) can be retrieved from the transducer M through projection operations.

[8]Any subset of the cross-product (including the empty set) between two sets, $A \times B = \{(a,b) \mid a \in A, b \in B\}$, is a binary relation over $A \times B$

[9]where the superscript n denotes n-repetitions of a given symbol, and $*$ denotes an infinite repetition of a given symbol

Operation	*Representation*
Union	$R(M_k) = R(M_i) \cup R(M_j)$
Complementing	Generally not closed
Difference	Generally not closed
Concatenation	$R(M_k) = R(M_i)R(M_j)$
Kleene-star	$R(M_k) = R(M_i)^*$
Intersection	Generally not closed
Cross product	$R(M_k) = L(A_i) \times L(A_j)$
Composition	$R(M_k) = R(M_i) \circ R(M_j)$

Table 5.3: Operations that result in regular relations

We can also combine machines in parallel. Using definitions for the finite-state machines $M_i = (\Sigma, \Gamma, Q_i, \delta_i, \omega_i)$ and $M_j = (\Sigma, \Gamma, Q_j, \delta_j, \omega_j)$, we obtain $M_k = (\Sigma, \Gamma, Q_i \times Q_j, \delta_k, \omega_k)$ where

$$\begin{aligned} \delta_k((p,q),\sigma) &= \{(p^{'},q^{'}) \mid \delta_i(p,\sigma) = p^{'} \ \wedge \ \delta_j(q,\sigma) = q^{'} \ \wedge \ \exists\, \omega_k((p,q),\sigma)\} \\ \omega_k((p,q),\sigma) &= \{\sigma' \mid \omega_i(p,\sigma) = \sigma' \ \wedge \ \omega_j(q,\sigma) = \sigma'\} \end{aligned} \tag{5.34}$$

Parallel connection is associative and commutative and is only defined if all machines are ϵ-free[10], and the number of resulting states is in practice less than the product of the number of states of the component machines, because many states are undefined or unreachable.

5.4.2 Transformations of Categories

This section is concerned with showing that any category can be transformed into another category (including itself). Our inputs and outputs are regular languages (the set of all strings recognized by some finite state automaton), and the space of all possible combinations consists of each and every input matched to each and every output. If a machine is presented with any pair of input and output strings and eventually halts in an accepting state, then we have shown that any category can indeed be transformed into any other category. The proof that transformation of one *category* into another category is computable, is based on the construction of an appropriate transducer. To show that a transducer can be created that recognizes $L(A_1) \times L(A_2)$ relations, we use the following reasoning: as our starting point, we use definitions of $L(A_1)$and $L(A_2)$ to define the transducer T. We then show that this transducer has the property that it maps every word in $L(A_1)$ to a word in $L(A_2)$. We do this by showing that T is in a final state if and only if both automata representing $L(A_1)$ and $L(A_2)$ are also in final states. We conclude that a transducer T that computes any valid input/output pair exists, unless $L(A_1)$ or $L(A_2)$ are themselves poorly defined.

[10]Machines that are ϵ-free do not have transitions without a labelling symbol, as would be the case in a spurious transition

Theorem 5.4.1. *Given any category $c \neq \emptyset$, there exists a finite state transducer T that maps it to a subset of C_L: $\forall c \in C_L\ \exists T \mid c' = T(c),\ c' \subseteq C_L$. The computability of this mapping follows directly from the existence of the transducer T.*

Proof. Representing *categories* as languages (equation 5.29), we need to show that $L(A_1) \times L(A_2)$ relations are recognized by some transducer T. In this case, $L(A_1)$ is the language of the automaton $A_1 = (\Sigma, Q_1, q_1, F_1, \delta_1)$ representing the *category* c and $L(A_2)$ is the language of the automaton $A_2 = (\Sigma, Q_2, q_2, F_2, \delta_2)$ representing the *category* c'. We set $T = (\Sigma \cup \{\epsilon\}, Q_1 \times Q_2, (q_1, q_2), F_1 \times F_2, \delta_T)$. The transition function δ_T is defined as:

$$\delta_T((q_1, q_2), a, b) = \delta_1(q_1, a) \times \delta_2(q_2, b) \tag{5.35}$$

where $q_1 \in Q_1; q_2 \in Q_2; a, b \in \Sigma \cup \{\epsilon\}$, and we need to show that that for any pair (u, v) where $u, v \in \Sigma^*$, T is in a final accepting state $(f_n, f_m) \in F_1 \times F_2$ if and only if machines A_1 and A_2 are in accepting states f_n and f_m respectively. To achieve this, we must show that the following equation holds:

$$\hat{\delta}_T((q_1, q_2), x, y) = \hat{\delta}_1(q_1, x) \times \hat{\delta}_2(q_2, y) \tag{5.36}$$

By induction on the number of transitions (or length of input string if the transducer is deterministic) we have:

Basis: $\hat{\delta}_T((q_1, q_2), \epsilon, \epsilon) = \delta_T((q_1, q_2), \epsilon, \epsilon) = \delta_1(q_1, \epsilon) \times \delta_2(q_2, \epsilon) = \hat{\delta}_1(q_1, \epsilon) \times \hat{\delta}_2(q_2, \epsilon)$;

Induction: $\hat{\delta}_T((q_1, q_2), ua, vb) = \delta_T(\hat{\delta}_T((q_1, q_2), u, v), a, b)$ using equation 5.28; $\delta_T(\hat{\delta}_T((q_1, q_2), u, v), a, b) = \delta_T(\hat{\delta}_1(q_1, u) \times \hat{\delta}_2(q_2, v), a, b)$ is the inductive step from equation 5.36; $\delta_T(\hat{\delta}_1(q_1, u) \times \hat{\delta}_2(q_2, v), a, b) = \delta_1(\hat{\delta}_1(q_1, u), a) \times \delta_2(\hat{\delta}_2(q_2, v), b)$ using equation 5.35; and finally, $\delta_1(\hat{\delta}_1(q_1, u), a) \times \delta_2(\hat{\delta}_2(q_2, v), b) = \hat{\delta}_1(q_1, ua) \times \hat{\delta}_2(q_2, vb)$ using equation 5.28. The transducer T is therefore in accepting state if and only if both A_1 and A_2 are also in accepting states. □

5.4.3 Transformations of Catsets and USLs

Since catsets are sets of categories and USLs are sets of catsets, we use Theorem 5.4.1 to show that the transformation of catsets and USLs is also computable..

Theorem 5.4.2. *Given any catset $\kappa_i \neq \emptyset$, there exists a finite state transducer T that maps κ_i to some catset κ_o. The computability of this mapping follows directly from the existence of the transducer T.*

Proof. The transformation of a catset κ_i into a catset κ_o implies that each *category* in the catset $c_i \in \kappa_i$ is transformed into some *category* $c_o \in \kappa_o$. Both c_i and c_o represent languages, and we can represent catsets in terms of languages: $L_\kappa = \bigcup_{n=0,\ c_n \in \kappa}^{m} c_n$. Setting L_{κ_i} and L_{κ_o} to be languages of the catsets κ_i and κ_o respectively, where both L_{κ_i} and L_{κ_o} are subsets of L_{IEML} by equation 5.9, we first obtain two automata that recognize L_{κ_i} and L_{κ_o} and then we follow

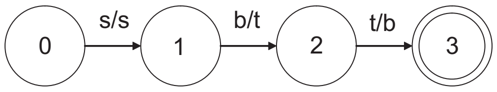


Figure 5.2: Seme exchange (*attribute* $\Leftrightarrow$ *mode*) operation for the *category* $\{s, b, t\}$

the same reasoning as in Theorem 5.4.1 to show the existence of a transducer T, which encodes the relation between L_{κ_i} and L_{κ_o}. $\square$

Theorem 5.4.3. *Given any USL $u_i \neq \emptyset$, there exists a finite state transducer T that maps u_i to some USL u_o. The computability of this mapping follows directly from the existence of the transducer T.*

Proof. To show the existence of a transducer T that performs the mapping $u_o = T(u_i)$, theorem 5.4.2 is used: since a transducer that performs the mapping $\kappa_o = t(\kappa_i)$ exists for every $\kappa_i \in u_i$ and $\kappa_o \in u_o$, and transducers are closed under union operation (see table 5.3), then $T = \bigcup_{n=0}^{6} t_n$, where t_n is the transducer that performs the mapping $\kappa_o^n = t_n(\kappa_i^n)$. $\square$

5.4.4 Examples of Transformational Operations

This section discusses operations on IEML categories and is grounded in the result of Theorem 5.4.1. Although the theorem ensures that there is a machine for a particular operation, it does not detail how to create the said machine. The following examples show machines performing a given operation.

5.4.4.1 Seme exchange

The following machine performs a simple seme exchange on a *category*:

$$\begin{array}{l} \Sigma = \Gamma = \{s, b, t\} \\ Q = \{0, 1, 2, 3\} \\ Q_0 = \{0\} \\ \Delta = \{(0, s, s, 1), (1, b, t, 2), (2, t, b, 3)\} \\ F = \{3\} \end{array} \tag{5.37}$$

The machine defined by equation 5.37 takes as input a *category* $\{s, b, t\}$, from which it outputs the $\{s, t, b\}$ *category*, as shown in figure 5.2.

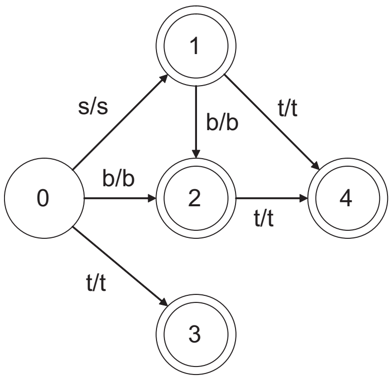


Figure 5.3: Powerset operation for the *category* $\{s, b, t\}$

5.4.4.2 Powerset operation

Considering the *category* Power Set function, we can readily construct a finite state machine using definition 5.4.2 to represent it:

$$\begin{array}{l} \Sigma = \Gamma = \{s, b, t\} \\ Q = \{0, 1, 2, 3, 4\} \\ Q_0 = \{0\} \\ \Delta = \{(0, s, s, 1), (0, b, b, 2), (0, t, t, 3), (1, b, b, 2), (1, t, t, 4), (2, t, t, 4)\} \\ F = \{1, 2, 3, 4\} \end{array} \tag{5.38}$$

The machine defined by equation 5.38 produces/recognizes the set $\bar{s}$:

$$\bar{s} = \{\{s\}, \{b\}, \{t\}, \{s, b\}, \{s, t\}, \{b, t\}, \{s, b, t\}\} \tag{5.39}$$

and is shown in figure 5.3. In that figure, the state 0 is the starting state and the double circles represent final - or accepting - states. As an example, if the input to the machine is s while the machine is in the starting state, the machine will output s and move to state 1. In this state it only accepts b and t which move the machine to states 2 and 4, respectively. At this point the machine is said to have produced/recognized the set $\bar{s}$:

$$\bar{s} = \{\{s\}, \{s, b\}, \{s, t\}\} \tag{5.40}$$

5.4.4.3 Partition operation

The partition operation requires a basis, a role address and a partitioner. In mathematical terms, the basis is a set S (possibly of other sets), the role address indicates on which part of any $s \in S$ the partition occurs, and the partitioner specifies the exact partition to perform. This operation maps sets of layer l to sets of layer l, $S^l \to P^l$ such that $s \in S \iff s \in \bigcup_i p_i$ where $p_i \in P$ and $\bigcap_i p_i = \emptyset$.

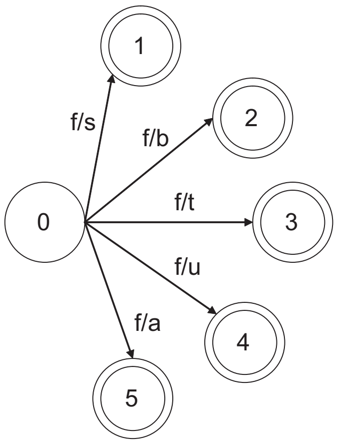


Figure 5.4: Partition operation for the *category* $f = \{u, a, s, b, t\}$

A finite state machine that represents this operation is a non-deterministic transducer. For example, one partition of a *category*

$$f = \{u, a, s, b, t\} \tag{5.41}$$

is the set $\{\{s\}, \{b\}, \{t\}, \{u\}, \{a\}\}$, which is represented by the following machine:

$$\begin{gathered}
\Sigma = \{s, b, t, u, a\} \\
\Gamma = \{s, b, t, u, a\} \\
Q = \{0, 1, 2, 3, 4, 5\} \\
Q_0 = \{0\} \\
\Delta = \{(0, f, s, 1), (0, f, b, 2), (0, f, t, 3), (0, f, u, 4), (0, f, a, 5)\} \\
F = \{1, 2, 3, 4, 5\}
\end{gathered} \tag{5.42}$$

The machine defined by equation 5.42 produces/recognizes the set $\bar{s}$:

$$\bar{s} = \{\{s\}, \{b\}, \{t\}, \{u\}, \{a\}\} \tag{5.43}$$

and is shown in figure 5.4. In that figure, state 0 is the starting state and the double circles represent final - or accepting - states.

5.4.4.4 Rotation operation

The rotation operation requires a basis, a role address and a rotor. In mathematical terms, the basis is a set S, the role address indicates on which part of any $s \in S$ the rotation occurs, and the rotor specifies the exact rotation to perform. In general, this operation maps sets of layer l to sets of layer l, $S^l \to P^l$.

A finite state machine that represents this operation is a non-deterministic transducer. One rotation of a *category* $\{s, b, t\}$ is the set $\bar{s}$:

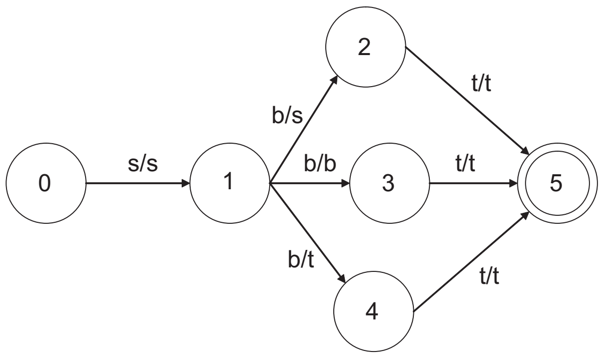


Figure 5.5: Rotation operation for the *category* $\{s, b, t\}$ with $\{\{s\}, \{b\}, \{t\}\}$ rotor

$$\bar{s} = \{\{s, s, t\}, \{s, b, t\}, \{s, t, t\}\} \tag{5.44}$$

assuming the rotor is given by $\{\{s\}, \{b\}, \{t\}\}$ and the role address is the attribute, which is represented by the following machine:

$$\begin{gathered}\Sigma = \Gamma = \{s, b, t\} \\ Q = \{0, 1, 2, 3, 4, 5\} \\ Q_0 = \{0\} \\ \Delta = \{(0, s, s, 1), (1, b, s, 2), (1, b, b, 3), (1, b, t, 4), (2, t, t, 5), (3, t, t, 5), (4, t, t, 5)\} \\ F = \{5\}\end{gathered} \tag{5.45}$$

Note that the role address decides where the non-determinism will occur: in the given example, it is on the second state. The machine defined by equation 5.50 produces/recognizes the set $\bar{s}$:

$$\bar{s} = \{\{s, s, t\}, \{s, b, t\}, \{s, t, t\}\} \tag{5.46}$$

and is shown in figure 5.5. In that figure, state 0 is the starting state and the double circle represents the final - or accepting - state.

5.4.4.5 Supertriplication operation

The supertriplication operation is the union of all applicable triplication operations (see equation 5.3). Once transducers for triplication operations are defined, a union operation on those transducers is performed, which results in yet another transducer (transducers are closed under union operation, see Table 5.3). The supertriplication operation is thus also represented by a (composite) transducer.

5.4.4.6 Superselection operation

For any given layer, by assigning a particular order to sets we can perform a superselection operation on those sets. In mathematical terms, given two ordered sets A and B, where $|A| = |B|$, then $A \times B \rightarrow C$, where $\forall c_{ij} \in C$, $a_i \in c_{ij} \wedge b_j \in c_{ij}$. The superselection operation results in a regular IEML matrix.

5.4.4.7 Matrix concatenation operation

Matrices can be concatenated together and are obtained by the union of the sets that represent the matrices. Note that this is not the same as a union operation on the underlying sets. If we assume that $A \times B \rightarrow C$, $A^{'} \times B^{'} \rightarrow C^{'}$, then in the former case we have $C \cup C^{'} = C^{''}$, whereas in the latter we would obtain $(A \cup A^{'}) \times (B \cup B^{'}) = C \cup C^{'} \cup A \times B^{'} \cup A^{'} \times B$.

5.5 Relations Model

5.5.1 Generalities

Relations are modelled with graphs. A graph is a tuple $G = (V, E)$ of two sets, such that E is the set of two-element subsets of V: $E \subseteq [V]^2$. For instance, if $V = \{a, b, c, d\}$ then $[V]^2 = \{\{a, b\}, \{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}, \{c, d\}$, and E can be represented by $\{\{a, b\}, \{b, c\}\}$. In general, the number of members in the set $[V]^k$ (k-element subsets of V) is calculated by combining n elements picked k at-a-time without repetition, and can be expressed as $\frac{n!}{k!(n-k)!}$.

If the vertices are the *categories* of a set, and the edges represent relations between those *categories*, then the resulting graph models the network of relations between *categories* of a particular set. Similarly, if the vertices represent sets of *categories*, and the edges represent relations between those sets, then the resulting graph models the network of relations between sets of *categories.*

The number of vertices in a graph determines its order and is represented as $|G|$, while the number of edges is represented as $||G||$. Graphs can be finite, denumerable or infinite, depending on their order. Edges connect vertices, and two vertices a, b are said to be adjacent if the edge $ab \in G$.

5.5.1.1 Relation similarity and nesting

Two graphs are isomorphic if a bijection $\phi : V \rightarrow V'$ exists with $ab \in E \iff \phi(a)\phi(b) \in E'$ for all $a, b \in V$. In essence, a mapping must exist between the vertices of the two graphs such that all the edges are preserved. The existence of isomorphism signifies that the structure of a particular relation is similar to the structure of another relation.

A subgraph G' of G is denoted $G' \subseteq G$ and requires that $V' \subseteq V$and $E' \subseteq E$. For a subgraph G', if it contains all edges $ab \in E$ with $a, b \in V'$ then it is an induced subgraph of G and is denoted $G' = G[V']$. Subgraphs efficiently represent the nesting of relations.

5.5.1.2 Connectedness of relations

The degree $d(v)$ of a vertex v is the number of edges at v. The minimum and maximum degrees (not the order!) of a graph are represented as $\delta(G) \triangleq min\{d(v)|v \in V\}$ and $\Delta(G) \triangleq max\{d(v)|v \in V\}$ respectively, while the average degree of a graph is given by: $\frac{1}{|V|}\Sigma_{v \in V} d(v)$. If all the vertices have the same degree, then the graph is regular (e.g., a cubic graph, for degree 3).

5.5.1.3 Hierarchical relations and trees

A relation may represent some hierarchy, which is represented as a *tree.* A graph G that does not contain any cycles is a forest. If such a graph is furthermore connected, then it is called a tree. Some general properties of trees (as follows) hold for any type of semantic relation that is represented as a tree:

Proposition 5.5.1. *For a graph G, if any two vertices are connected by a unique path, then G is a tree.*

Proof. The graph is connected since there is a path between any two vertices. A cycle requires two paths between the same vertices. A unique path between any two vertices in a graph, which therefore implies that there are no cycles in the graph. □

Proposition 5.5.2. *For a tree T, the path connecting any two vertices of T is unique.*

Proof. All vertices are connected, and since there are no cycles in T, the path connecting any two vertices must be unique. □

Proposition 5.5.3. *If a new edge joins two vertices in a tree T, then a cycle is formed.*

Proof. Since T is a tree, then there is a unique path from c to u and from c to v, and $\forall\, c, u, v \in T$. If a new edge joins u and v, a cycle $c \dots uv \dots c$ is formed. □

Proposition 5.5.4. *A tree T with n vertices has $n - 1$ edges.*

Proof. (informal) A tree with one vertex has no edges. Adding a second and subsequent vertex to the tree results in the addition of exactly one edge: it cannot be less than one because the graph would not be connected, and it cannot be more because a cycle would be created. □

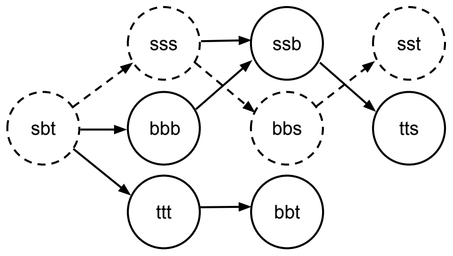


Figure 5.6: Graphical representation of some relation. The highlighted portion is a path between *sbt* and *sst* and can be viewed as a *category* $\{sbt, sss, bbs, sst\}$.

5.5.1.4 Basic relation operations

The union of two graphs G and G' is, by definition, $G \cup G' \triangleq (V \cup V', E \cup E')$, i.e., the union of the graphs' respective vertices and edges. Similarly, the intersection of two graphs is, by definition, $G \cap G' \triangleq (V \cap V', E \cap E')$. The difference between two graphs is obtained by deleting all common vertices and their incident edges. Many other operations on graphs exist. For example, if G and G' are disjoint graphs (where $V \cap V' = \emptyset$), then $G * G'$ is the graph obtained from $G \cup G'$ after joining all vertices in G with all vertices in G'. Another example is the complement on G, denoted $\bar{G}$, which is the graph on V with $\bar{E} = [V]^2 \backslash E$.

5.5.1.5 Relation paths

A path is a graph $P = (V, E)$ such that

$$V = \{x_1, x_2, \ldots, x_n\} \tag{5.47}$$

and

$$E = \{x_1x_2, x_2x_3, \ldots x_{n-1}x_n\} \tag{5.48}$$

and can be represented as a sequence of vertices, $x_1x_2 \ldots x_n$. A graph G is connected if any two of its vertices are linked by some path in G. Paths can represent IEML categories (see figure 5.6) and such a representation provides an interesting manner to describe catsets and USLs.

5.5.1.6 Relation cycles

Given a path $P = x_1x_2 \ldots x_{n-1}$ where $n \geqslant 3$, the graph $C = P + x_{n-1}x_1$ is a cycle (two edges connecting the same vertices in an undirected graph is treated as one edge). The minimum and maximum length of a cycle contained in a graph G are called the girth and the ,circumference respectively. In the case of a tree (see section 5.5.1.3), the girth is set to ∞ and the circumference to 0.

5.5.1.7 Relation distances

The distance between two vertices x and y of a graph G, $d_G(x, y)$ is the length of the shortest path between x and y. If no such path exists, the distance is ∞.

The eccentricity of any vertex x is the maximum distance between x and any other vertex in the graph G. The diameter of a graph is given by the maximum of its vertex eccentricities, $diam_G = max_{y \in V(G)}(d_G(x, y))$ and its radius by the minimum of its vertex eccentricities, $rad_G = min_{x \in V(G)}(diam_G)$. When a vertex in a graph is equally or less distant to any other vertex than to the graph's radius, then it is central in that graph. The radius and minimum degree of a graph that represents any type of semantic relation can be used for classification purposes. For example, a graph G of radius no more than k and of degree $d \geq 3$, representing any type of semantic relation, has less than $\frac{d}{d-2}(d-1)^k$ vertices.

Proof. Let z be a central vertex in G, and D_i the set of vertices of G at a distance of i from z. The total number of vertices in G is given by $V(G) = \bigcup_{i=0}^{k} D_i$. The following is true: for $i = 0$, $|D_0| = 1$ since it contains only the vertex z, and for $i = 1$, and $|D_1| \leqslant d$ since it cannot exceed the maximum degree of the graph. For $i \geqslant 1$, the following holds: $|D_{i+1}| \leqslant (d-1)|D_i|$ since each vertex in D_{i+1} shares at least one edge with a vertex in D_i. Thus we obtain $|D_{i+1}| \leqslant d(d-1)^i$, $\forall i < k$. The number of vertices in G is therefore $|V(G)| \leqslant \sum_{i=0}^{k} |D_i| = 1 + \sum_{i=0}^{k-1} |D_{i+1}|$. The sum $s_k = \sum_{i=0}^{k-1}(d-1)^i$ can be rewritten as $s_k = 1 + (d-1) + (d-1)^2 + \ldots + (d-1)^{k-1}$ and can be subtracted from $s_k(d-1)$ which gives $s_k(d-1) - s_k = (d-1)^k - 1$. Solving for s_k gives $\frac{(d-1)^k-1}{d-2}$. By inserting this in the previous equation, we obtain $|V(G)| \leqslant 1 + d\frac{(d-1)^k-1}{d-2} < \frac{d}{d-2}(d-1)^k$, since $\frac{d}{d-2} > 1$ $\square$

5.5.2 Relations and Semantic Graphs

Semantic graphs are a representation of the semantic relations found between IEML expressions: categories, catsets and USLs.

5.5.2.1 Linear order relations

For any graph $G = (V, E)$ that is a tree (see section 5.5.1.3) and where $\forall v \in V$, the degree (see section 5.5.1.2) always conforms to $0 < d(v) \leq 2$ and describes a path (see section 5.5.1.5). The edges E of these graphs define a linear-order relation between *categories*, catsets and USLs. Many such graphs can exist, depending on the exact ordering criteria.

5.5.2.2 Set-subset relations

This type of relation occurs only between *categories* of the same layer. In the general case all *categories* that are in a set-subset relation are given by $C_L^r \subseteq C_L \times C_L$, where C_L^r is given by:

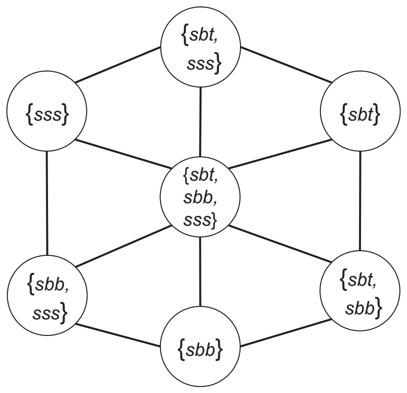


Figure 5.7: Graphical representation of set-subset relation for the *category* $\{sbt, sbb, sss\}$.

$$C_L^r = \{\{c_i, c_j\} \mid c_i, c_j \in C_L, c_i \subseteq c_j\} \tag{5.49}$$

The graph $G = (V, E)$ is given by:

$$\begin{aligned} V &= C_L \\ E &= C_L^r \end{aligned} \tag{5.50}$$

Set-subset relations can also be constructed for particular cases by obtaining the powerset of that *category* of interest that defines the vertices V, and applying the formula in equation 5.50. A representation for the *category* $\{sbt, sbb, sss\}$ is shown in figure 5.7.

Set-subset relations are also applicable to catsets and USLs: two different catsets are in a set-subset relation if and only if there exists a *category* in both catsets that is in a set-subset relation; two different USLs are in a set-subset relation if and only if there exists a catset in both USLs that is in a set-subset relation.

In the general case, all catsets that are in a set-subset relation are given by $\kappa_L^r \subseteq C_L \times C_L$ where κ_L^r is given by:

$$\kappa_L^r = \{\{\kappa, \kappa_j\} \mid \exists c_i \in \kappa_i, \exists c_j \in \kappa_j, \{c_i, c_j\} \in C_L^r\} \tag{5.51}$$

Similarly, all USLs that are in a set-subset relation are given by $\mathcal{USL}^r \subseteq \mathcal{USL} \times \mathcal{USL}$ where $\mathcal{USL}^r$ is given by:

$$\mathcal{USL}^r = \{\{u_i, u_j\} \mid \exists \kappa_i \in u_i, \exists \kappa_j \in u_j, \{\kappa_i, \kappa_j\} \in \kappa_L^r\} \tag{5.52}$$

5.5.2.3 Symmetric relations

This type of relation occurs between *categories* c and c' of the same layer and of the same cardinality (*categories* must contain the same number of sequences) whenever there exist: an automaton (see section 5.4.1.1) A that recognizes some sub-sequence $s \in L_{IEML}$ in both c and c', and a transducer (see section 5.4.1.3) T that computes some relation cTc' (transforms c into c'). The automaton A describes a similarity at the level of symbolic arrangement (or syntactic level) and is a necessary condition for the assertion of a semantic symmetry, while the transducer T describes some semantic invariance combined to a semantic variation. Both the A and T machines can be derived from the specific *directory* (see section 5.6.4.5) that contains categories of interest.

Catsets are in a symmetric relation if and only if there exists a *category* in both catsets that are in a symmetric relation, and USLs are in a symmetric relation if and only if there exists a catset in both USLs that are in a symmetric relation.

5.5.2.4 Etymological relations

An etymological relation is in general a relation between a *category* at layer n and one of its semes at layer n–n', where $1 \leq n' < n$. Depending on the directory that contains categories of interest, an etymological relation may or may not be present. The graph representation of these relations has a tree structure.

5.6 Semantic Circuits

5.6.1 Paradigmatic Characters and Sequences

Using the alphabet defined by equation 5.17, we distinguish ten symbols in $\bar{\Sigma}$: $\{T\}$, $\{B\}$, $\{S\}$, $\{A\}$, $\{U\}$, $\{E\}$, $\{S,B,T,U,A,E\}$, $\{S,B,T,U,A\}$, $\{S,B,T\}$ and $\{U,A\}$, which symbols are a subset of the category language alphabet (see equation 5.17). These symbols, and only these symbols, are called paradigmatic characters and form the paradigmatic alphabet $\bar{\Sigma}_P$. Use of the $\bar{\Sigma}_P$ alphabet allows us to differentiate paradigmatic sequences among the sequences of the category language (see section 5.2.4). IEML sequences are obtained from paradigmatic sequences by performing a Cartesian product between all the sets of the paradigmatic sequence. For instance, the paradigmatic sequence $\{S,B,T\}\{U,A\}\{E\}$ gives the IEML set of sequences $\{SUE,\ SAE,\ BUE,\ BAE,\ TUE,\ TAE\}$. Using the definition in 5.1, paradigmatic sequences for $0 \leq l \leq 6$ are given by the general expression:

$$L^P_{IEML} = \{s \in \bar{\Sigma}^*_P \mid |s| = 3^l\} \tag{5.53}$$

where $L^P_{IEML} \subseteq L_{IEML}$.

5.6.2 Paradigmatic Distance

Using Equation 5.22 we consider a set of functions $F = \{f_1, f_2, \cdots, f_n\}$ where

$$\forall f \in F,\ f : \mathcal{USL} \rightarrow \mathcal{USL} \tag{5.54}$$

and a directed graph $G_p = (\mathcal{USL}, E)$ where:

$$E = \{(\mu_i, \mu_o) \mid (\mu_i, \mu_o) \in [\mathcal{USL}]^2 \ \wedge \ \mu_o = f(\mu_i),\ f \in F\} \tag{5.55}$$

The *paradigmatic path* between any two nodes $a, b \in \mathcal{USL}$ of the graph G_p is a path on G_p (see Section 5.5.1.5), which leads to the following definition of a *paradigmatic distance:*

Definition 5.6.1. *The shortest path between any two vertices a, b of G_p is the paradigmatic distance between a and b.*

The shortest path can be calculated by algorithms presented in Section 5.7.2.

5.6.3 Semantic circuits

Semantic circuits S_ω are labelled and directed graphs $G = (V, E)$ where $V \subseteq L_{IEML}$ and all $e \in E$ are a ternary relation (s, m, d) where $(s, d) \in [V]^2$ and $m \in L_{IEML}$. By the following proof we can show that the set of all semantic circuits O_c forms a *groupoid,* capturing symmetries described in section 5.3.4.3:

Proof. The binary permutation operation $\otimes$ is partial[11], implying that it is defined for some, but not all, members of O_c. The following properties hold:

- $\forall a, b, c \in O_c,\ (a \otimes b) \otimes c \rightarrow a \otimes (b \otimes c)$
- $\forall a \in O_c,\ \exists a^{-1} \in O_c,\ a \otimes a^{-1} = i_a$
- $\forall a, b \in O_c,\ a \otimes b \rightarrow a \otimes b \otimes b^{-1} = a$
- $a \otimes a^{-1} \otimes b = b$; $\forall a \in O_c,\ a \otimes a^{-1} \in O_c$

These properties are necessary and sufficient to characterize a groupoid. □

A particularly important subset of semantic circuits is the subset that preserves the category structure (see section 5.3.4) given by equation 5.23. Consider a subset of O_c such that $\forall a \in O_c^l \subset O_c,\ s = m = d = 3^l,\ 0 \leq l \leq 6$. The collection of O_c^l for $0 \leq l \leq 6$, together with a collection of morphisms $M_c = \{m_c \mid m_c = (o_c^l \rightarrow o_c^{l+1});\ 0 \leq l \leq 5\}$ and the triplication function from equation 5.3 applied to (s, m, d), form the *category* $\mathbb{S}$. We then can show that the functor F from *category* $\mathbb{C}$ to $\mathbb{S}$ is a mapping from $\mathbb{C}$ to $\mathbb{S}$ such that: $a_c \in O_c :\ F(a_c) \in O_s$; $(x_c \rightarrow y_c) \in M_c :\ (F(x_c) \rightarrow F(y_c)) \in O_s$ where $\forall a \in O_c,\ F(1_a) = 1_{F(a)}$ and $\forall a, b \in M_c,\ F(a \circ b) = F(a) \circ F(b)$. This is fundamental, because it states that the *category* of IEML language can be mapped to the *category* of semantic circuits.

[11]Weisstein, Eric W. "Partial Function." From MathWorld, A Wolfram Web Resource. http://mathworld.wolfram.com/PartialFunction.html

5.6.4 Rhizomes

A rhizome $G_\rho(V, E)$ is a type of semantic circuit where $V = \bigcup_i V_i$ and $\forall m, n \in V_i$, $\exists(m, n) \in E \wedge \exists(n, m) \in E$. This type of semantic circuit is thus composed of fully-connected subgraphs, or *cliques*.

5.6.4.1 Serial rhizome

A serial rhizome is a rhizome $G_s = (V_s, E_s)$ with the following properties: G_s is a path (see section 5.5.1.5), $V_s \subseteq L^P_{IEML}$, $\forall v_i, v_j, v_k \in V_s$, $|v_i| = |v_j|$, and there is a binary relation $\leq$ on V_s such that: $v_i \leq v_j \wedge v_j \leq v_i \rightarrow v_i = v_j$, $v_i \leq v_j \wedge v_j \leq v_k \rightarrow v_i \leq v_k$, $v_i \leq v_j \vee v_j \leq v_i$.

5.6.4.2 Etymological rhizome

An etymological rhizome is a rhizome $G_e = (V_e, E_e)$ with the following properties: G_e has a tree structure (see section 5.5.1.3), $V_e \subseteq L^P_{IEML}$, $\forall e_i, e_j \in E_e$, $e_i \neq e_j$, $e_i \in \texttt{substance} \cup \texttt{attribute} \cup \texttt{mode}$ of equations 5.6, 5.7 and 5.8.

5.6.4.3 Taxonomic rhizome

A taxonomic rhizome is a rhizome $G_t = (V_t, E_t)$ with the following properties: G_t is a tree (see section 5.5.1.3), $V_t \subseteq L^P_{IEML}$ and $\forall e_i \in E_t$, $e_i = \{(v_i, v_j) \mid \forall j, v_i = \bigcup_j v_j \ \wedge \ \bigcap_j v_j = \emptyset\}$.

5.6.4.4 Paradigms

A paradigm $\mathbb{P}$ is the result of a union, intersection or symmetric difference of taxonomic, etymological or serial rhizomes (see section 5.5.1.4).

5.6.4.5 Dictionary

A dictionary $\mathbb{D}$ is a paradigm where vertices have a one-to-one mapping to natural languages.

5.7 Quantitative Criteria for Semantic Circuits

5.7.1 Structural Similarity Criterion

5.7.1.1 Generalities

Definition 5.7.1. *Matrix $\boldsymbol{A}$ is a two-dimensional array of objects of the same class over a field*[12] Ψ*:*

$$\boldsymbol{A} = [a_{ij}] = \begin{bmatrix} a_{11} & \cdots & a_{1j} & \cdots & a_{1N} \\ \vdots & & \vdots & & \vdots \\ a_{i1} & \cdots & a_{ij} & \cdots & a_{iN} \\ \vdots & & \vdots & & \vdots \\ a_{M1} & \cdots & a_{Mi} & \cdots & a_{MN} \end{bmatrix}, \tag{5.56}$$

$$\forall \left\{ i = \overline{1 \ldots N}, j = \overline{1 \ldots M}; M, N \in \mathbb{Z} \right\} : a_{ij} \in \Psi \therefore \boldsymbol{A} \in \Psi^{N \times M}, \tag{5.57}$$

where M, N are called matrix dimensions.

As long as the matrix entries a_{ij} have operations of addition and multiplication defined on field Ψ, we can define matrix addition and multiplication over matrices of compatible dimensions as follows.

Definition 5.7.2. *For $\boldsymbol{A} \in \Psi^{N \times M}$ and $\boldsymbol{B} \in \Psi^{N \times M}$:*

$$\boldsymbol{A} + \boldsymbol{B} = [a_{ij} + b_{ij}] \,. \tag{5.58}$$

Definition 5.7.3. *For $\boldsymbol{A} \in \Psi^{N \times M}$, $\boldsymbol{B} \in \Psi^{M \times K}$ and $\boldsymbol{C} \in \Psi^{N \times K}$, $\boldsymbol{C} = \boldsymbol{AB}$ means that*

$$[c_{ij}] = \left[\sum_{m=1}^{M} a_{im} b_{mj} \right]. \tag{5.59}$$

It can be shown that matrix multiplication defined this way satisfies the usual multiplication properties of associativity [WEI 1999]:

$$(\mathbf{AB})\,\mathbf{C} = \mathbf{A}\,(\mathbf{BC})\,, \tag{5.60}$$

and of both left and right distributivity:

$$(\mathbf{A} + \mathbf{B})\,\mathbf{C} = \mathbf{AC} + \mathbf{BC} \tag{5.61}$$

$$\mathbf{C}\,(\mathbf{A} + \mathbf{B}) = \mathbf{CA} + \mathbf{CB}. \tag{5.62}$$

[12]A field is a ring (see 5.3.1) equipped with the operations of addition, subtraction, multiplication and division that satisfy the axioms of closure, associativity, commutativity, identity, inverse, and distributivity [ROT 2003].

As for commutativity, even if **BA** exists, case commutativity generally does not hold [STR 2009]:

$$\mathbf{AB} \neq \mathbf{BA}. \tag{5.63}$$

We can now define two additional useful concepts: identity matrix and matrix inverse.

Definition 5.7.4. *An identity matrix $\boldsymbol{I}$ is a matrix that satisfies Proposition 5.3.2:*

$$\forall \boldsymbol{A} \in \Psi^{N\times M} \exists \boldsymbol{I} \in \Psi^{M\times M} : \boldsymbol{A}\boldsymbol{I} = \boldsymbol{A} \tag{5.64}$$

Lemma 5.7.1. *If Ψ is equipped with a scalar null element $\nvdash$:*

$$\forall a \in \Psi \; \exists \nvdash \in \Psi : a + \nvdash = a \wedge \nvdash \cdot a = \nvdash, \tag{5.65}$$

and a scalar identity element $\nVdash$:

$$\forall a \in \Psi \; \exists \nVdash \in \Psi : \nVdash \cdot a = a, \tag{5.66}$$

the identity matrix $\boldsymbol{I} = [\iota_{ij}] \in \Psi^{M\times M}$ for a matrix $\boldsymbol{A} \in \Psi^{N\times M}$ can then be computed as

$$\iota_{ij} = \begin{cases} \nVdash & \Leftrightarrow i = j \\ \nvdash & \Leftrightarrow i \neq j \end{cases} \tag{5.67}$$

Proof. By substituting properties (5.65) and (5.66) into equation (5.59) according to rule (5.67), we obtain

$$\mathbf{AI} = \left[\sum_{m=1}^{M} a_{im}\iota_{mj} \right] = [a_{ij}\iota_{jj}] = [a_{ij} \cdot \nVdash] = [a_{ij}] = \mathbf{A} \tag{5.68}$$

□

It is trivial to also show that identity matrix multiplication is one of those special cases where the commutativity property does indeed hold as long as the underlying scalar multiplication on Ψ is commutative:

$$\mathbf{AI} = \mathbf{IA} = \mathbf{A}. \tag{5.69}$$

For the sake of simplicity, we will only define the inverse of a square matrix $A \in \Psi^{N\times N}$. For a generalized treatment, the interested reader can refer to the Moore-Penrose pseudoinverse in [GOL 1996], for example.

Definition 5.7.5. *Matrix $\boldsymbol{A}^{-1} \in \Psi^{N\times N}$ is called an inverse of $\boldsymbol{A} \in \Psi^{N\times N}$ if the following property holds:*

$$\boldsymbol{A}\boldsymbol{A}^{-1} = \boldsymbol{A}^{-1}\boldsymbol{A} = \boldsymbol{I}. \tag{5.70}$$

If $\boldsymbol{A}^{-1}$ does not exist, $\boldsymbol{A}$ is called a singular matrix.

Yet another useful concept in matrix algebra is a permutation matrix, which enables us to identify matrices whose differences can be expressed as permutations of their columns and rows.

Definition 5.7.6. *Matrix $\boldsymbol{P}$ is called a* permutation *matrix if it satisfies the following conditions:*

1. *$\boldsymbol{P}$ is a square binary matrix: $\boldsymbol{P} \in \mathbb{B}^{N\times N}$, $\mathbb{B} := \{0, 1\}$.*
2. *$\boldsymbol{P}$ has exactly one identity entry for each row and each column, and null entries elsewhere: $\forall i, j = \overline{1 \ldots N} : \sum_{i=0}^{N} p_{ij} = 1 \wedge \sum_{j=0}^{N} p_{ij} = 1$.*

5.7.1.2 Adjacency matrix

Matrices provide a convenient representation for the manipulation and study of graphs. One such matrix is the *adjacency matrix* of a graph representing its connectivity [CHA 1984].

Definition 5.7.7. *For a graph $G = (V, E)$, $|E| = N$ its adjacency matrix is a unique matrix $\boldsymbol{A} \in \mathbb{R}^{N\times N}$, where $\mathbb{R}$ is the set of real numbers. For a simple graph G the adjacency matrix $\boldsymbol{A}$ is computed as follows:*

$$a_{ij} = \begin{cases} |v_i v_j| & \Leftrightarrow i \neq j \\ \kappa |v_i v_i| & \Leftrightarrow i = j \end{cases} \tag{5.71}$$

where $|v_i v_j|$ is the number of edges from vertex i to vertex j, and $\kappa = 1$ for directed graphs and $\kappa = 2$ otherwise.

5.7.1.3 Graph isomorphism

Graphs that have the same structure and differ only in insignificant details, such as numbering on vertices and edges, are studied using the notion of *isomorphism* [DIE 2005].

Definition 5.7.8. *Two graphs G_1 and G_2 with adjacency matrices $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ are called* isomorphic *if and only if there exists a permutation matrix $\boldsymbol{P}$ such that*

$$\boldsymbol{P}\boldsymbol{A}_1\boldsymbol{P}^{-1} = \boldsymbol{A}_2 \tag{5.72}$$

This definition of isomorphism naturally lends itself to testing if a particular graph transformation is isomorphic. If the transform operator generates a matrix that satisfies Definition 5.7.6, then the matrix is a permutation matrix and condition (5.72) is satisfied automatically. On the other hand, proving a negative proposition (i.e., that two graphs G_1 and G_2 are not isomorphic) requires a proof of a negative proposition that no such permutation exists that transforms G_1 into G_2. A solution to this dilemma will be provided below in Section 5.7.1.4 (Lemma 5.7.3).

5.7.1.4 Graph spectral theory

Definition 5.7.8, although mathematically rigorous, is not very practical. More practical ways of studying graph isomorphisms, among other graph properties, are among the many applications of the spectral graph theory. Spectral graph theory studies properties of graphs in relation to the eigenvalues [GOL 1996] of the matrices that completely describe the graph, such as the adjacency matrix, the distance matrix, or the admittance matrix (also called Laplacian matrix, [WEI 1999]).

Definition 5.7.9. *A scalar $\lambda \in \mathbb{C}$ is an* eigenvalue *of a square matrix $\boldsymbol{A}$ if it satisfies the following equation:*

$$(\boldsymbol{A} - \lambda \boldsymbol{I})\, \boldsymbol{q} = 0, \tag{5.73}$$

where $\boldsymbol{A} \in \mathbb{R}^{N\times N}$, $\boldsymbol{I} \in \mathbb{R}^{N\times N}$ and $\boldsymbol{q} \in \mathbb{C}^{N\times 1}$. The column vector $\boldsymbol{q}$ associated with that eigenvalue is called an eigenvector.

In scalar terms, system (5.73) has N equations for $N+1$ unknowns, so its solution $(\lambda, \mathbf{q})$ is not necessarily unique. Rather, it has up to N distinct solutions, which form multisets $\{\lambda_1, \dots, \lambda_N\}$ and $\{\mathbf{q}_1, \dots, \mathbf{q}_N\}$.

Definition 5.7.10. *A multiset of eigenvalues $\{\lambda_1, \dots, \lambda_N\} \in \mathbb{C}$ of a matrix $\boldsymbol{A} \in \mathbb{R}^{N\times N}$ is called* spectrum *of $\boldsymbol{A}$.*

Lemma 5.7.2. *If $\boldsymbol{A} \in \mathbb{R}^{N\times N}$ is symmetrical, then all of its eigenvalues are real numbers:*

$$\forall (i,j) = \overline{1 \dots N} : a_{ij} = a_{ji} \;\Leftrightarrow\; \{\lambda_1, \dots, \lambda_N\} \in \mathbb{R} \tag{5.74}$$

and have an orthogonal set of eigenvectors:

$$\forall (i,j) = \overline{1 \dots N} : \left\langle \boldsymbol{q}^{(i)}, \boldsymbol{q}^{(j)} \right\rangle \neq 0 \Leftrightarrow i = j \tag{5.75}$$

Corollary 5.7.1. *Undirected graphs have real spectra (immediately follows from (5.71)).*

Definition 5.7.11. *Two graphs G_1 and G_2 are called* isospectral (cospectral) *if and only if they have identical spectra $\Lambda(G_1) = \Lambda(G_2)$.*

Lemma 5.7.3. *Non-isospectral graphs are necessarily not isomorphic:*

$$\Lambda(G_1) \neq \Lambda(G_2) \Rightarrow \nexists \boldsymbol{P} : \boldsymbol{P}\boldsymbol{A}_1\boldsymbol{P}^{-1} = \boldsymbol{A}_2 \tag{5.76}$$

Matrix spectra can be efficiently computed using the QR algorithm for relatively small matrices or the Lanczos algorithm for large sparse matrices [GOL 1996]. Together with the result of Lemma 5.7.3 this provides a practical way of testing for the absence of isomorphism between different graphs.

Spectral graph theory is currently a very active area of research in mathematics, particularly for undirected graphs. Interested readers are further referred to [CVE 1997] for an extensive review of the field.

Algorithm 5.1 Computing distance matrix for unweighted graphs

```
input : Adjacency matrix $\mathbf{A} \in \mathbb{Z}^{N\times N}$
output: Distance matrix $\mathbf{D} \in \mathbb{Z}^{N\times N}$
begin
    $\mathbf{B} \leftarrow \mathbf{I}_N \in \mathbb{Z}^{N\times N}$
    $\mathbf{D} \leftarrow \infty_N \in \mathbb{Z}^{N\times N}$
    for $n = 1$ to $N$ do
        $\mathbf{C} \leftarrow \mathbf{BA}$
        for $i = 1$ to $N$ do
            for $j = 1$ to $N$ do
                if $c_{ij} > 0$ and $b_{ij} = 0$ then
                    $d_{ij} \leftarrow n$
```

5.7.2 Shortest Path Criterion

5.7.2.1 Shortest path problem for unweighted graphs

In exploring graphs it is often interesting to compute the length of the shortest path between a certain pair of vertices or, more generally, between all pairs of vertices.

Definition 5.7.12. *A graph $G = (V, E)$ is called* unweighted *if the distances between every pair of adjacent vertices (i, j) are set to be the same $\forall (i, j) = \overline{1 \ldots N}$: $d_{ij} = const$. Otherwise, we call the graph* weighted.

Without loss of generality, it is often convenient to assume the basic distance in an unweighted graph to be 1.

For unweighted graphs, this problem can be solved in a rather elegant way by computing the power series $\mathbf{A}^n$ of the adjacency matrix. $\mathbf{A}^n$ has an interesting property as its entries $a_{ij}^{(n)}$ are equal to the number of paths of length n between vertices i and j [CHA 1984]. The following algorithm computes the *distance matrix* $\mathbf{D} \in \mathbb{Z}^{N\times N}$ that contains distances d_{ij} between vertices i and j.

It follows rather obviously that algorithm 5.1 has a computational complexity of N matrix multiplications or $O(N^3)$. It works for both directed and undirected graphs, and computes a complete set of all-pairs shortest path lengths measured in terms of number of steps between the vertices.

5.7.2.2 Generalized shortest path problem

Nonetheless, semantic circuits (Section 5.6.3) can be much more accurately represented by weighted graphs where weights l_{ij} are assigned in inverse proportion to the importance of connection between particular USLs. In order to solve the all-pairs shortest path problem for weighted graphs, let us define the fundamental distance matrix $\mathbf{D}^{(0)} \in \mathbb{R}^{N\times N}$ that contains the lengths of all the paths of first order, where *arc* denotes a function that checks whether two vertices are adjacent.

Algorithm 5.2 Computing distance matrix for weighted graphs

```
input  : $\mathbf{P}^{(0)} \in \mathbb{R}^{N\times N}$, $\mathbf{D}^{(0)} \in \mathbb{R}^{N\times N}$ according to (5.77)
output: $\mathbf{D}^{(N)}$, $\mathbf{P}^{(N)}$
begin
    for $i, j = \overline{1 \dots N}$ do
        if $i = j$ then
            $p_{ij}^{(0)} \leftarrow$ NaN  (IEEE not-a-number value)
        else
            $p_{ij}^{(0)} \leftarrow i$
    $k \leftarrow 1$   repeat
        for all $i, j = \overline{1 \dots N}$ do
            $d_{ij}^{(k)} \leftarrow \min\left[\left(d_{ij}^{(k-1)}\right), \left(d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)\right]$   if $d_{ij}^{(k)} \neq d_{ij}^{(k-1)}$ then
                $p_{ij}^{(k)} \leftarrow p_{kj}^{(k-1)}$
            else
                $p_{ij}^{(k)} \leftarrow p_{ij}^{(k-1)}$
        $k \leftarrow k + 1$
    until $k > N$;
```

Definition 5.7.13.

$$\forall (i,j) = \overline{1 \dots N} : d_{i,j}^{(0)} := \begin{cases} 0 & \Leftarrow i = j \\ l_{ij} & \Leftarrow \exists\ arc(i,j) \\ \infty & \Leftarrow \nexists\ arc(i,j) \end{cases} \tag{5.77}$$

The following iterative algorithm [ATA 1998] is attributed to Floyd and Warshall and is widely used in numerous applied fields, such as comparative genetics, circuit analysis and operations research [LAR 1997].

On completion, algorithm 5.2 returns matrices $\mathbf{D}^{(N)}$ and $\mathbf{P}^{(N)}$ with $d_{ij}^{(N)}$ containing the length of the shortest path between vertices i and j, while matrix $\mathbf{P}^{(N)}$ contains information that makes it possible to trace these shortest paths. The computational complexity of the algorithm is equal to $N + 1$ iterations of $\mathrm{O}(N^2)$ each or $\mathrm{O}(N^3)$ in total.

Index

Bibliography

[ARZ 1999] Arzi-Gonczarowski Zippora, "Perceive this as that – analogies, artificial perception, and category theory". *Annals of Mathematics and Artificial Intelligence*, 26(1-4): 215–252, 1999.

[ATA 1998] Atallah Mikhail J. (eds.), *Algorithms and Theory of Computation Handbook.* CRC Press: Boca Raton, FL, 1998.

[BAC 2000] Bacry Henri (preface by Alain Connes), *La symétrie dans tous ses états.* Vuibert: Paris, 2000.

[BAR 2002] Barabasi Albert Laszlo, *Linked, the New Science of Networks.* Perseus: Cambridge, MA, 2002.

[BEA 2002] Béal Marie-Pierre and Olivier Carton, "Determinization of transducers over finite and infinite words". *Journal of Theoretical Computer Science*, 289(1): 225–251, October 2002.

[BUT 1991] Butler Gregory, "Fundamental algorithms for permutation groups". *Lecture Notes in Computer Science*, 559, 1991.

[CHA 1984] Chartrand Gary, *Introductory Graph Theory.* Dover Publications: New York, NY 1984.

[CHO 1963] Chomsky Noam and Marcel P. Schützenberger, "The algebraic theory of context-free languages", in Braffort, P. and D. Hirschberg (eds.): *Computer Programming and Formal Languages.* North Holland, Amsterdam, 118-161, 1963.

[CVE 1997] Cvetkovic Dragos, Rowlinson Peter, and Simic Slobodan, *Eigenspaces of Graphs.* Cambridge University Press: Cambridge, UK 1997.

[DIE 2005] Diestel Reinhard, *Graph Theory.* Springer-Verlag: Heidelberg, 2005. On-line at: http://diestel-graph-theory.com

[GIL 1962] Gill A. *Introduction to the Theory of Finite-State Machines.* McGraw-Hill Book Company: NY 1962.

[GOL 1996] Golub Gene H. and Charles F. van Loan, *Matrix Computations.* 3rd Edition. The John Hopkins University Press: Baltimore, MD 1996.

[HEN 1968] Hennie Frederick C., *Finite-State Models for Logical Machines.* John Wiley & Sons, Inc.: NY and London 1968.

[HOL 1991] Holzmann Gerard J., *Design and Validation of Computer Protocols.* Prentice Hall: Upper Saddle River, New Jersey 1991.

[HOP 2001] Hopcroft John E., Motwani Rajeev and Ullman Jeffrey D., *Introduction to Automata Theory, Languages, and Computation.* 2nd edition. Addison-Wesley,: Boston 2001.

[KAP 1994] Kaplan Ronald M. and Martin Kay, "Regular models of phonological rule systems". *Computational Linguistics*, 20(3): 331–378, 1994.

[KEL 1994] Kelly Kevin, *Out of Control. The New Biology of Machines, Social Systems and the Economic World.* Addison Wesley: NY, 1994.

[KLE 1956] Kleene Stephen, *Representation of Events in Nerve Nets and Finite Automata*, Princeton University Press: Princeton, N.J., 3–42, 1956.

[LAR 1997] Larson Richard C. and Odoni Amedeo R., *Urban Operations Research.* Chapter 6.2: "Travel Distances on Networks", Massachusetts Institute of Technology: Cambridge, MA, 1997-99. On-line at http://web.mit.edu/urban_or_book/www/book/

[LEV 2011] Lévy, Pierre, *The Semantic Sphere. Computation, cognition and the information economy.* ISTE Wiley, New-York and London, 2011.

[ROC 1997] Roche Emmanuel, "Compact factorization of finite-state transducers and finite-state automata". *Nord. J. Comput.*, 4(2): 187–216, 1997.

[ROT 2003] Rotman Joseph J. *Advanced Modern Algebra.* Prentice Hall: Englewood Cliffs, NJ 2003.

[STR 2009] Strang Gilbert, *Introduction to Linear Algebra.* 4th Edition. Wellesley-Cambridge Press: Wellesley, MA 2009.

[WEI 1999] Weinstein Eric W., *CRC Concise Encyclopedia of Mathematics.* CRC Press: Boca Raton, FL 1999.