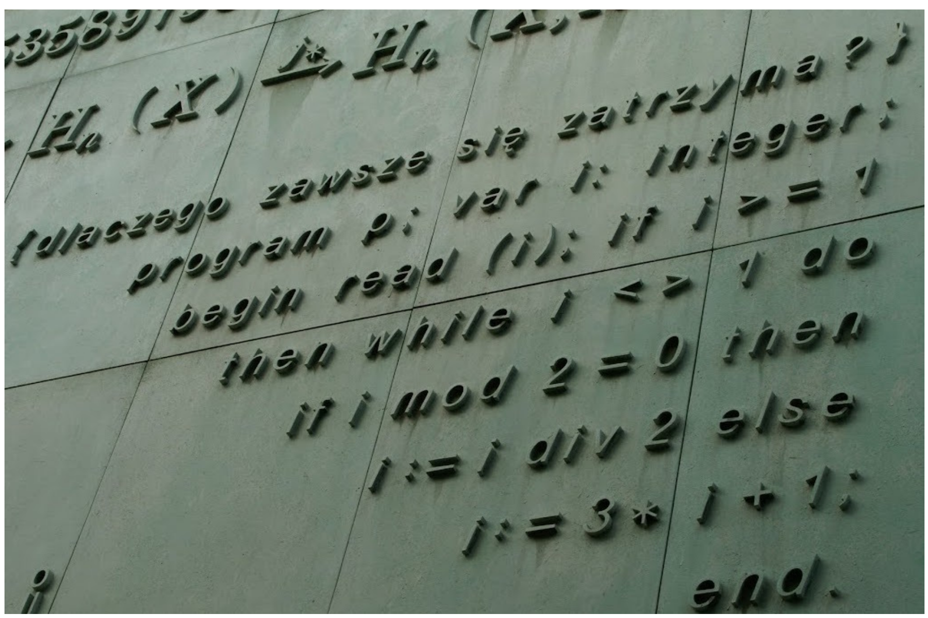
INFORMATICS FOR BEGINNERS

Laymens First Introduction
A School Teachers Compendium

Dines Bjørner

August 19, 2023: 11:09 am



Dines Bjørner
Institute of Mathematics and Computer Science
Technical University of Denmark
Fredsvej 11, DK-2840 Holte, Denmark

History:

- This document was started early July 2023.
  I plan to write this book over the next two years: **"my 2 year project !"**
  I am almost daily writing a bit here, a bit there — after a previous night's thoughts!
  It has become my only and first hobby!

Editorials:

- As of August 19, 2023: 11:09 am this is [still] a **draft**.
- As a draft, its distribution will be very limited.
- Seemingly "advanced-level" texts of many chapters serve as a *depository*.
  That depository material is expected to be worked into "introductory-level" text, i.e., "severely" cut and made "accessible" for laymen!
- The form and contents of this book emerges, little-by-little.
  Presently I have arranged for many chapters, undoubtedly too many.
  I expect, as I "progress", that some will be removed.
- A series of chapters, maybe not all, to begin and end with the same items:
  Motivation, Study, etc., respectively Summary and Exercises.
  Very few exercises have so far been inserted.
  There therefore is a Solutions chapter, i.e., Chapter B.
  Very little "appears" there, at the moment (12.Aug.2023).

Preface

This book introduces the reader to the field of *Informatics:* **Informatics** is, to this author, confluence of **mathematics**, of the **computer & computing sciences**, of the **domain science & engineering** [as espoused in this book], and, to some extent also *requirements engineering and software design*.[1]

This book also introduces the reader to MoLa: an abstract, yet computable, Modeling Language[2] The reader will learn to read and understand MoLa programs and models, that is, how to understand essential features of **computer programs** and **domain models** – the latter such as, for example, *banking*, *road transport*, *railways*, *retailing*, and even such phenomena as *canals !*

We make a distinction between programs and models. Programs are intended for interpretation by, i.e., "execution" on, computers.[3] Models are [abstract program-like] specifications intended primarily, in this book, for reading, understanding and possibly also experimentation, by humans !

This book is intended to as a first introduction to *domain models & computing* for *laymen*, i.e., for people with at least 8 years of schooling and otherwise curious of the world around them, and, more specifically, for *teachers and students at primary-to-middle school*

---

[1] We emphasize that this is our delineation of the term 'informatics'. There are others: Edinburgh University defines informatics as follows: *Informatics studies the representation, processing, and communication of information in natural and engineered systems. It has computational, cognitive and social aspects. The central notion is the transformation of information - whether by computation or communication, whether by organisms or artifacts.* [`https://www.ed.ac.uk/files/atoms/files/what20is20informatics.pdf`]

`[Wikipedia]` informs us: *Informatics is the study of computational systems.*[1][2] *According to the ACM Europe Council and Informatics Europe, informatics is synonymous with computer science and computing as a profession,*[3] *in which the central notion is transformation of information.*[1][4] *In other countries, the term "informatics" is used with a different meaning in the context of library science, in which case it is synonymous with data storage and retrieval.[5] [`https://en.m.wikipedia.org/wiki/Informatics`]*

[1] "What is Informatics ?" University of Edinburgh.

[2] "INFORMATICS" — Bedeutung im Cambridge Englisch Wörterbuch"; dictionary.cambridge.org (in German).

[3] "Are We All In The Same Boat ?" ACM & Informatics Europe.

[4] "What is Informatics ?" - Definition from Techopedia. Techopedia.com. October 2014.

[5] Wellisch, Hans (1972-07-01). "From Information Science to Informatics: a terminological investigation". Journal of Librarianship.

[2] MoLa is a pared-down version of `RSL`: the `R`AISE `S`pecification `L`anguage [47], where `RAISE` is the `R`igorous `A`pproach to `S`oftware `E`ngineering method [48]. MoLa omits, among other things, `RSL`'s "object-oriented" specification structuring constructs: *Scheme, Class, Object*.

[3] MoLa is, however, not a computer programming language. We do not provide an computer interpreter [1] or compiler [2] for MoLa. Some readers may themselves develop and provide such, hopefully *public domain*, software, i.e., "for free", that enable computer assisted calculations in MoLa.

[1] In computer science, an interpreter is a computer program that directly executes instructions written in a programming or scripting language, without requiring them previously to have been compiled into a machine language program `[Wikipedia]`.

[2] A compiler is a special program that translates a programming language's source code into machine code, byte-code or another programming language `[Wikipedia]`.

levels interested in teaching basic matters related to computing.

For these latter, one may expect these college teachers and their candidates to write specific, primary, respectively middle school textbooks on computing. Those school textbooks are then intended to teach the basics of computing seen from the view of mathematical abstractions.

An Essence

This book emphasizes that the readers learn how to read and understand basic properties of the everyday world that surrounds them – not the [equally exciting] world of [intricate, clever, complex] algorithms performed by computers in calculating properties of that world. Also we shall, presently, not cover such quintessential aspects of computing as the correctness of software nor the complexity of algorithms.

The study of this book is to be done without computers! Yes, indeed! It is the concepts that underlie models and computing, not the information technology (IT) implements, the hardware, [computers, say in the form of laptops, data communication, as illustrated by The Internet, storage, etc.] that are at the center of our concern.

• • •

This author has, for many years, been quite unhappy of the ways in which we teach computing in schools.

As we grow up we learn to *speak*. The ability to understand human *speech* and to *speak* is not inherited. We painstakingly *learn* it.

As we grow up we learn to *read* and *write*. The ability understand human *written text* and to *write* is not inherited. Humans "invented" *languages* and *expressing language text in spoken word.* We painstakingly *learn* it.

In school we learn, first *reckoning*[4], then *mathematics*[5]. We learn to read and understand mathematics before we learn to 'mimic' mathematics in the sense of learning to solve basically trivial problems. They are trivial in the sense that the problem poser knows the solution beforehand. We learn about *mathematical theories* and their application in solving problems. We do not learn to "create" new mathematics, new theories. At universities we learn how mathematicians "invented", came across, studied and thus created new mathematics.

Likewise for *physics*. In school we did not learn to "make new" physics. We learn about *theories of physics: mechanics, electricity, etc.* and their application in solving problems. [We did learn, however, how physicists came about and created new physics![6]]

[4]Reckoning: *calculating* "with" actual *numbers*, first the whole, positive, "natural" ones, then also the negative ones, the *integers*, moving on, slowly, to *rationals* and *reals* [the rationals + the irrationals], *imaginary*, *transcendental*, etc.

[5]Mathematics, here in the sense of "lifting" from concrete numbers to symbolic, i.e., named numbers: $x, y, \ldots$. Then "lifting" to logic, sets, etc., etc.

[6]Galileo's [1564–1642] observing the "similar" falls of light and heavy masses. Newton's [1643–1727] apple dropping on his head, Etc.

When first introducing computers and their programs in schools, schools, in most instances, introduced programming, not to first understand theories of programs.

This book turns all this “up-side–down” ! First on reading and understanding programs for computers; then on reading and understanding models of primarily man-made domains;

For actual programmers a main concern is that of efficiency of computer programs: faster, using less storage, etc. That is a “material concern”. For us, to read and understand programs and models a main concern is ease of understanding, elegance, “beauty” ! [43, Edsger W. Dijkstra]. That is an “intellectual concern!”

Holte August 19, 2023: 11:09 am

Contents

Chapter 0

Introduction

Contents

Motivation: Education

In this chapter we "entertain" the reader with a palette of widely different topics! It is meant as a gentle, hopefully not too distracting survey of a rather wide set of concepts – all deemed necessary for the full enjoyment of this oeuvre! [1]

Section **0.7.3** (pages 8–10) should be studied before remaining chapters are read!

> **Study Hint:** *This chapter can be studied at Your leisure. It calls for so-called "arm-chair" reading, with a cup of good, warm, West Lake[2] Long Jing (Dragon Well) green tea and biscuits!*

0.1 Domains

Christians take comfort in:

```
In the beginning was the Word, and the Word was with God, and the Word was God.
                                            The same was in the beginning with God.
All things were made by him; and without him was not any thing made that was made.
                                    In him was life; and the life was the light of men.
                  And the light shineth in darkness; and the darkness comprehended it not.
                                    There was a man sent from God, whose name was John.
The same came for a witness, to bear witness of the Light, that all men through him
                                                                      might believe.
```

[1] We 'label' this box *Education* – in the sense of the German word *Bildung*, and the Danish word *Dannelse*, that is, in th sense of achieving insight, becoming wiser, rather than becoming more proficient in certain skills!

[2] 'West Lake' ('Xi Lake') is in the city of HangZhou, ZheJiang Province, China

```
He was not that Light, but was sent to bear witness of that Light.
That was the true Light, which lighteth every man that cometh into the world.
He was in the world, and the world was made by him, and the world knew him not.
```
[*King James I Bible, From the Gospel of John, 1:1–10. 1611*[3]]

We shall, perhaps profanely, interpret this *bringing together* two such diverse concepts as *"The Word"* and *the world & universe* around us: *"All things were made by Him"*, to justify this book's emphasis on *domains* and a formal domain description *language*.

Therefore Chapter **1**'s introduction to the concept of **domains** [*All things ... made*]. And therefore most remaining chapters' introduction to a formal domain *model* description language [*The Word*].

Definition 1 **Domain:** *By a domain we shall understand a rationally describable segment of a discrete dynamics fragment of a human assisted reality, i.e., of the world that we daily observe. It includes its* **endurants**, *i.e., solid and fluid entities of* **parts** *and* **living species**, *and* **perdurants** ∎

Endurants are either natural ["God-given"] or artefactual ["man-made"]. and may be considered atomic or compound parts, or, as in this book, further unanalysed living species: plants and animals – including humans.

Perdurants are here considered to be actions, events and behaviours.

Chapter **1**, therefore, is *the first, major chapter* of this book.

0.2 The Concepts and Practice of Programs and Models

Caveat:

We shall use the term 'model' in a specific sense: Foremost it is used in the sense of **domain models.** That is where we start. From the "outside" ! From the man-made artefactual world around us as it is embedded in the natural world.

And we shall use the term 'program' to mean: **possibly computable program code**, thereby also working from the "inside-out !"

That is: We do not start with the fact of computers, but with the fact of "real world" domains.

This book is to serve as an introductory, i.e., first text, on the *concepts and practices* of **domain modelling**, as well as on the *concepts and practices* of **computable programs.**

It is addressed to *laymen*[4] and *primary* and *middle school teachers* and their *teachers* at teachers' colleges – and all other people in-between !

Definition 2 **Concepts of Computing:** [5] *By the concepts of computing we shall include the ideas of • computation; • formal texts prescribing computations; • a syntactical text*

[3] John took his cue from Genesis:1–5

[4] Layman: a person without professional or specialized knowledge in a particular subject

[5] Concerning enumeration of definitions: Appendix chapter, Chapter A, brings an extensive list of concept definitions, numbered from 1 up. The present definition of *Concepts of Computing* therefore, enumeration-wise, "starts" where Chapter A "ends" !

being subject to [machine, i.e., computer] interpretation; • [a] specification; • a program; • programming; and • abstraction ▪

As this book will show, there are many more computing concepts.

Definition 3 Practices of Computing: *By the practices of computing we shall include those of • abstraction; • conceptualisation (of a problem, an exercise or its solution); • narration (of a problem, an exercise or its solution); •programming; and • divide-and-conquer* ▪

As this book will show, there are many more computing practices.
The *concepts & practices* of *domain modelling* will occupy most of this book. Basic domain modelling concepts are introduced in Chapter **1**.
It is intended for both self-study and for teachers college classes in computing. From the present book such teachers should be able to, themselves, develop proper class material for primary and middle school students.

0.3 Teaching Tools: Speech and Writing

This book outlines a teaching of the practical fundamentals of computing which does not make use of computers! We suggest that teaching the practice of computing can be done by speech and writing. Writing as in writing on a [black, white, green (glass fiber)] board and on paper! Speech as spoken by the teacher and the class students.

We cannot, of course, avoid, that some teachers may supplement their use of speech and writing by the use of computers in the class room. That is, computer tools may be developed which allow examples "being computed"! In this day-and-age of, so-called, AI, such tools may make use of AI.[6]

0.4 Didactics

Definition 4 Didactics: *A didactic method (Greek: $\delta\iota\delta\alpha\sigma\kappa\epsilon\iota\nu$, "to teach") is a teaching method that follows a consistent scientific approach or educational style to present information to students* [`https://en.wikipedia.org/wiki/Didactic_method`].

The didactics of this book is based on the following principles:

- Programs are mathematical objects.
- Although quite deep mathematical theories can be expounded, relevant to the understanding of programs and programming, one need only a modicum of logic and discrete mathematics to get along with the basics of programming.

[6] We however, find that, however, "perfect" the so-called Man-Machine-Interface, MMI, the interface between the machine, i.e., the computer hardware, and the student – however "perfect" – it may be, that it does, invariably, stand in the way of pedagogy!

- A didactics is that of not introducing concepts before constituent concepts – i.e., concepts in terms of which the concept being introduced – are defined. Thus the concept of *sets* is introduced *before* the concept of *numbers*!

- A final, major didactics is also the following: Rather than focusing on computing, we focus on specification of possibly computable problems, more specifically that of the specification of **domains.**

  That is: analysing and describing domains will be more dominant than analysing and describing [possibly intricate and beautiful] algorithms.

0.5 Problem Solving

Programming is a human *process, following some procedure, for solving problems*. What do we mean by *tool, solving* and *problem*? For *process*, see below, Definition 7 Sect. **0.6**. For *tools*, see below, Definition 9 on the next page Sect. **0.6**. By solving we mean such things as *getting answers* to *questions.* Variations on the theme: *problem* are: (i) *posing a question* – whose possible answer is not immediately obvious – represents *formulating a problem*; (ii) *an inquiry starting from given conditions to investigate or demonstrate a fact, result, or law* – represents another facet of the concept of *problem*.

0.6 Method & Methodology

Definition 5 **Method:** *By a* **method** *we shall understand a set of*

- **principles** *and* **procedures**

  for selecting and applying a set of

- **techniques** *and* **tools**

to a problem in order to achieve an orderly construction of a **solution** ■

Definition 6 **Principles:** *By a* **principle** *we mean: a proposition or value that is a guide for behavior or evaluation* `[Wikipedia]`*, i.e., code of conduct* ■

Definition 7 **Procedure:** *By a* **procedure** *we mean: instructions or recipes, a set of commands that show how to achieve some result, such as to prepare or make something* `[Wikipedia]`*, i.e., an established way of doing something* ■

Definition 8 **Technique:** *By a* **technique** *we mean: a technique, or skill, is the learned ability to perform an action with determined results with good execution often within a given amount of time, energy, or both* `[Wikipedia]`*, i.e., a way of carrying out a particular task* ■

Definition 9 Tool: *By a* **tool** *we mean: a tool is an object that can extend an individual's ability to modify features of the surrounding environment [Wikipedia]* ■

Definition 10 Formal Method: *By a* **formal method** *we shall understand a method*

- *whose principles include that of considering its artifacts as mathematical quantities, of abstraction, etc.;*
- *whose decisive procedures include that of*
  - *the sequential analysis and description of first endurants, then perdurants, and,*
  - *within the analysis and description of endurants, the sequential analysis and description of first their external qualities and then their internal qualities,*
  - *etc.;*
- *whose techniques include those of specific ways of specifying properties; and*
- *whose tools include those of one or more* **formal languages** ■

0.7 Languages

Natural Language is the principal method of human communication, consisting of words used in a structured and conventional way and conveyed by speech, writing, or gesture.

0.7.1 Definitions of Languages

MoLA is the formal language of this book, the object language.

Definition 11 Language: *By a* **language** *we shall here understand a set of strings of characters, i.e., sentences, sentences which are structured according to some*

- **syntax**[7], *i.e.,* **grammar**,
- *are given meaning by some* **semantics**[8], *and*
- *are used according to some* **pragmatics**[9] ■

[7]By *syntax* we shall mean the arrangement of elements (e.g., words or parts) and their composition (e.g., phrases or composite parts) to create well-formed structure (e.g., sentences or parts) in a language or model. [By words and phrases we mean those of a (written/spoken) laguages; and by parts we mean those of a domain model.]

[8]By *semantics* we shall mean we shall mean the meaning of a syntactic element (word, phrase, text or part). [By words and phrases we mean those of a (written/spoken) laguages; and by parts we mean those of a domain model.]

[9]By *pragmatics* we shall mean we shall mean what a speaker/writer/domain-modeler implies and a listener/reader infers based on contributing factors like the situational context, the individuals' mental states, the preceding dialogue, and other factors.

Definition 12 Formal Language: *By a* **formal language** *we shall here understand a language*

- *whose* **syntax** *and* **semantics** *can both be expressed* **mathematically** *and*
- *about whose sentences one can* **rationally reason** *(argue, prove)* **properties** ■

0.7.2 Two Languages

Two language will be at play in this book:

- *the object language*, here MoLa, and
- *the language of the observer's presentation.*

  We also refer to the language of the observer as the, or a, *meta language*.

We quote from Kleene [62, Chapter 1, § 1] – relevant to Chapter **2**:

> *When we are studying logic, the logic we are studying will pertain to one language, which we call the object language, because this language (including its logic) is an object of our study. Our study of this language and its logic, including our use of logic in carrying out the study, we regard as taking place in another language, which we call the observers language. Or we may speak of the object logic and the observers logic.*

Similar remarks can be made for all chapters' presentation of material.

0.7.2.1 The Object Language

The object language of this book is MoLa.

We shall slowly build up, i.e., introduce, this language. Basic elements of MoLa derive from logic (hence Chapter **2**), sets (hence Chapter **3**), numbers and numerals (hence Chapter **4**), functions and types (hence Sect. **6.1**), Cartesians (hence Chapter **8**), lists (hence Chapter **10**), maps (hence Chapter **11**), etcetera.

0.7.2.2 The Observer's Language

The observer's language, i.e., the meta language "borrows" from many fields: Foremost from English. Then from mostly discrete mathematics, not that of MoLa, but that of the discrete mathematics You learn in school and at colleges and universities. And finally from pictorial, graphical languages. Only discrete mathematics has a formal, if not always a logic, foundation.

0.7.2.3 Applicative MoLa

$\boxed{\text{to be written}}$

0.7.2.4 Imperative MoLa

$\boxed{\text{to be written}}$

0.7.2.5 Concurrency MoLa

$\boxed{\text{to be written}}$

0.7.3 Formal Expression Evaluation

This section has direct bearing, i.e., is of importance, to a proper understanding of the concept of formal language, and is hence of importance to grasping the semantics of MoLa.

0.7.3.1 Formal Expressions

From early school years You have come across such *formal expressions* as:

- $a^2 + b^2 = c^2$ – Pythagoras'[10] theorem, right-angled triangle
- $a * x^3 + b * x^2 + c * x + d = 0$ – ordinary polynomial in x of degree 3.
- $E = mc^2$ – Albert Einstein's[11] 1905 equation relating Energy (of a body of material) to the $m$ass (of that material) and the square of the speed of light, c^2.

First: they are *formal* since their *syntax* is expressed in a precisely defined language, i.e., as here, that of mathematics. Secondly: they are *formal* since their *semantics* is well defined – also in ordinary mathematics and physics!

Definition 13 Formal Expression: *By a formal expression we shall here mean an expression which lends itself to formal evaluation, i.e., any kind of "calculation" that is systematic* ■

In this book we shall, again-and-again, introduce the formal expressions of MoLa.

0.7.3.2 Evaluation

Definition 14 Formal Evaluation: *By formal evaluation we mean a process whereby we — in a mathematically rigorous manner – find some property, for example the value — of an expression* ■

So far we have not really introduced the concept of MoLa expressions. Further on, i.e., immediately below, we shall introduce the

So, for the time being, kindly asking or Your patience, let us assume that $\mathcal{E}$ is a formal expression. As an example of such, let it range over

- constants,
- variables[12], and

[10] Greek: around 500 BC

[11] 18979–1955

[12] identifiers which denote values

- sub-expressions which are of the form
  - $\mathsf{E}_\ell\ \mathsf{O}_i\ \mathsf{E}_r$, or
  - $\mathsf{O}_p\ \mathsf{E}$,

  whee O_i are infix, dyadic, two argument operators, and O_p are prefix [or suffix], unary operators, and E_ℓ, E_r, E are appropriate expressions.

Evaluation now proceeds as follows:

- If the expression is a ***constant***, like **true, false**, $0, 1, ..., a, b, c, d$, then the denoted [constant] value is the result of the evaluation.
- If the expression is a ***variable***, like a, b, c, d in Pythagoras' theorem or x in the polynomial equation, then the denoted value – somehow "kept" in the environment of the evaluation – is the result of the evaluation.
- If the expression is an infix expression then the operation denoted by the infix operator is applied to the result of the evaluation of the two operand expressions. And
- If the expression is a prefix expression then the operation denoted by the prefix [or suffix] operator is applied to the result of the evaluation of the operand expression.

MoLa is a language built up around formal expressions.

0.7.3.3 Failed Evaluation: **chaos**

To express that some evaluation fails, for example the division of any number x by 0, we introduce the literal

- **chaos**.

chaos is a literal of MoLa.

Throughout this book we shall be concerned with proper forms of uses of MoLa in order to avoid failed evaluations.

0.7.4 MoLa Models

Although it is quite "early" in our presentation of a method for describing domains in MoLa and introducing [MoLa] programming ideas we shall, nevertheless, state the following:

Definition 15 Descriptions and Models: *A MoLa description is a set of* ***specification units*** ■

Definition 16 Specification Units: *The MoLa specification units are of the following kinds:*

- **type**,
- **value**,
- **axiom**,
- **variable**, *and*
- **channel** ■

The specification unit set is, for practical reasons, sequentially ordered, as is the text of this book.

We explain the role of specification units.

- **type** clauses introduce and define classes of values;
- **value** clauses introduce and define either specific or free-ranging values, as needed, of the MoLa specification; and
- **axiom** clauses state properties of values in the MoLa specification.

Type, value and axiom clauses make up the vast bulk of any MoLa specification.

We shall rarely have need for the additional MoLa specification units:

- **variable** clauses declare so-called 'assignable' variables, such as typically used in so-called *imperative programming*[13].
- **channel** clauses declare means of communication between MoLa behaviours.

0.7.5 Speech Acts

Definition 17 Speech Act: *Speech acts*[14] *are acts*[15] *that refer to the action performed by produced utterances. People can perform an action by saying something. Through speech acts, the speaker can convey physical action merely through words and phrases. The conveyed utterances are paramount to the actions performed* ■

We shall, going somewhat "outside" established convention wrt. 'speech acts', consider two forms:

- *speech acts* which **change** a "state of affairs", which moves You from one *state of* **proficiency**, say *capability*, to another, usually "improved" state; and
- *speech acts* which **change** Your *mood*, Your **insight**, *knowledge*.

In this book the text of some chapters predominantly aim at *proficiency*, typically in the method of programming in MoLa, while other chapters predominantly aim at *insight*, typically in understanding the basis for that method (and MoLa).

[13] Ordinary programming languages, from the 1954 `FORTRAIN`, via the 1960 `ALgol 60`, to todays `C, C`$^+$`,` `Java` and `Python` are all imperative programming languages.

[14] See also: `https://en.wikipedia.org/wiki/Speech_act`

[15] Acts are *things done; deeds, pretense*

0.8 Ontology and Taxonomy

to be written

The broader definition of ontology is:

Definition 18 **Ontology:** *By ontology we shall here mean: a set of concepts and categories in a subject area or domain that shows their properties and the relations between them* ■

Example 1 **Ontology:** We shall only be concerned with an ontology for domain description. See Fig. **1**.1 on page 27 ■

The broader definition of taxonomy is:

Definition 19 **Taxonomy:** *By a taxonomy we shall here understand the classification of something, in particular a specific domain* ■

Example 2 **Taxonomy:** A taxonomy for a simplification of a road transport domain is shown in Fig. **??** on page ?? ■[16]

more to come

0.9 Computer and Computing Science

Definition 20 **Computer Science:** *By computer science we shall mean the study and knowledge of the phenomena that "goes on, occur, inside" computing devices* ■

Definition 21 **Computing Science:** *By computing science we shall mean the study and knowledge of how to construct "those things" that "occur" within computers* ■

0.10 A Triptych of Software Development

The Triptych Dogma

In order to *specify* **Software**,
we must understand its requirements.
In order to *prescribe* **Requirements**
we must understand the domain
So we must **study, analyze** and **describe Domains**.

By a *domain* we shall understand a *rationally describable* segment of a *discrete dynamics* fragment of a *human assisted* reality, i.e., of the world ■

[16]**Editorial note:** To be inserted

0.10.1 Domains

Domains include **endurants**, i.e., *solid and fluid entities* of **parts** and **living species**, and **perdurants**. Perdurants are either *actions*, *events* and *behaviours*.

Endurants are either *natural* ["God-given"] or *artefactual* ["man-made"]. and may be considered *atomic* or *compound* parts, or, as in this book, further unanalyzed *living species:* **plants** and **animals** – including *humans*. Perdurants evolve around *states*. *States* are collections of parts. *Actions* potentially change states in a planned manner. *Events* surreptitiously change states. *Behaviours* are sets of sequences of actions, events and [*embedded*] behaviours.

It is a main characteristic of this book that it focuses very much on the presentation of material that enables the reader to model every-day domains such as mostly man-made systems, for example:

- **Banking:** By programming 'banking' we shall, for example, mean: *to create a model of bank customers, i.e., clients, of bank accounts, of deposits, withdrawals and loans, etc., and of credit/debit cards, etc.*
- **Road Transport:** By programming 'road transport' we shall, for example, mean: *to create a model of roads and automobiles, of traffic signals, routes along the roads, automobiles entering and leaving road intersections and street segments, etc.*
- **Retailing:** By programming 'retailing' we shall, for example, mean: *to create a model of customers, retailers, goods for sale, wholesalers/importers, etc.*
- **Railways:** By programming 'railways' we shall, for example, mean: *to create a model of rail nets, trains, time-tables, train rides, etc.*

but also [predominantly] natural domains, classical mathematical systems and basic computing systems:

- **Rivers and Canals:** By programming 'rivers and canals' we shall, for example, mean: *to create a model of rivers: their sources, confluence and deltas, and of canals: their networks, connection to rivers and the sea, locks, boat traffic, etc.*
- **Graphs, Trees:** By programming 'graphs and trees' we shall, for example, mean: *to create a model of these abstract notions, of edges and vertices, roots and leaves, paths, cycles, etc.*
- **Relational Databases:** By programming 'relational databases' we shall, for example, mean: *to create a model of relations and their tuples, and of their querying, i.e.,* `SQL`*, etc.*

0.10.2 Requirements

0.10.2.1 Some Definitions and Rules

Definition 22 Requirements, I: *By a [software] **requirements** we shall understand (cf., [256, IEEE Standard 610.12]): "A condition or capability needed by a user to solve a problem or achieve an objective"* ■

- The **Golden Rule** of requirements engineering: Prescribe only those requirements that can be objectively shown to hold for the designed software ■

Objectively shown means that the designed software can either be tested, or be model checked, or be proved (verified), to satisfy the requirements. Caveat: Since we do not illustrate formal tests, model checking nor theorem proving, we shall, alas, not illustrate adherence to this rule.

- An **Ideal Rule** of requirements engineering: When prescribing (including formalizing) requirements, also formulate tests and properties for model checking and theorems whose proof should show adherence to the requirements ■

The rule is labelled *ideal* since such precautions will not be shown in this book. The rule is clear. It is a question for proper management to see that it is adhered to. See the Caveat above.

- **Adequacy:** Make sure that requirements cover what users expect ■

That is, do not express a requirement for which you have no users, but make sure that all users' requirements are represented or somehow accommodated. In other words: the requirements gathering process needs to be like a "fine-meshed net": One must make sure that all possible stake-holders have been involved in the requirements acquisition process, and that possible conflicts and other inconsistencies have been obviated.

- **Implementability:** Make sure that requirements are implementable ■

That is, do not express a requirement for which you have no assurance that it can be implemented. In other words one must tacitly assume, perhaps even indicate, somehow, that an implementation is possible. But the requirements in and by themselves, may stay short of expressing such designs. Caveat: The domain and requirements specifications are, in our approach, model-oriented. That helps expressing "implementability".

Definition 23 Requirements, II: *By requirements we shall [further] understand a document which prescribes desired properties of a **machine:** what endurants the machine shall "maintain", and what the machine shall (must; not should) offer of functions and of behaviours while also expressing which events the machine shall "handle"* ■

Definition 24 Machine, II: *By a* machine *that "maintains" endurants we shall mean: a machine which, "between" users use of that machine, "keeps" the data that represents these entities* ∎

From earlier we repeat:

Definition 25 Machine, III: *By machine we shall understand a, or the, combination of hardware and software that is the target for, or result of the required computing systems development* ∎

So this, then, is a main objective of requirements development: to start towards the design of the hardware and software for the computing system.

Definition 26 Requirements, III: *To specify the machine* ∎

0.10.2.2 General

Domain models can be developed, final ones can be studied and enjoyed, without there being any thought of software developed for the domain!

But software for domains ought not, we almost "religiously" proclaim, be developed without first having [more-or-less] understood the domain, i.e., without having a domain model from which to start software development.

The link between a *domain description* and *domain software specification* is a *requirements prescription*.

Domain descriptions typically rely on abstractions for their expressiveness, and cover more aspects of the domain for which possible computing support may sought. As such, domain descriptions are typically expressed in terms of [elegant, short, concise] logical predicates, that, also typically, do not lend themselves to some form of "immediate" interpretation by computer. The role of a requirements prescription is to *"bridge the gap"* between "not obviously, nor usually, efficient" computing – and is to focus on those aspects of the domain for which computing is sought, and to also render that focus computable!

We shall not, in this book show how to "concert", how to transcendentally deduce, requirements prescriptions from domain descriptions.

Chapter 9 in [23] shows how to do that! And the three books: [8–16] shows further techniques and the transition from requirements prescriptions to software.

0.10.3 Software

Definition 27 Software: *By* software *we shall mean not just the code that, when submitted for execution on a computer, performs desired computations, but all the documentation that went into the development of that software: the underlying domain description, the ensuing requirements prescription, all the tests, checks and proofs of trustworthiness of these specifications, all the management plans, their follow-up, development histories, etc., etc.* ∎

0.10.4 Programs

Definition 28 **Program:** *A computer* program *is a sequence or set of instructions in a programming language for a computer to execute* ■

0.11 Computers

to be written

0.11.1 Hardware

to be written

0.11.2 Et cetera

to be written

0.12 Informatics & IT

- **Informatics:** We understand informatics as a confluence of mathematics, of the computer and computing sciences, of the domain science and engineering as espoused in this monograph, requirements engineering and software design.
- **IT – Information Technology:** We understand information technology as the confluence of nano physics, electronics, computers and communication (hardware), sensors, actuators, etc.
- **Two Universes:** Two diverse universes appear to emerge:

  Information Technology is, to this author, a *universe of both material quality and quantity*. It is primarily materially characterised, such as I see it, by such terms as *bigger, smaller; faster, slower; costly, inexpensive,* and *environment "friendly"*

  Informatics is, to this author, a *universe of intellectual quality*. As such it is primarily characterised, such as I see it, by such terms as *better, more fit for purpose, appropriate, logically correct* and *meets user expectations*.

0.13 Structure of Book

The[17] reader is urged to study, carefully, the overall table-of-contents, Pages v–viii, and the specific chapter table-of-contents that head each chapter!

[17] **Change:** *New text inserted: Structure of book section, Chapter* **0**

You may wish to skip the reading of the rest of this Structure of Book section in a first reading!

The book has **49** chapters! Some chapters address the issue of what should be learned and taught! They are usually short, some 10–15 pages at most. They could be taken, by school teachers, as the basis for a few class hours of teaching. The school teachers are then expected either, themselves, to develop teaching and problem exercise material for their classes – or to rely on primary-, middle- or high school text books provided by such, for example, teachers college professors who also could derive such text books from the present book. Other chapters address more "esoteric" issues.

to come

The book is structured in **XIV** parts!

- Part **I**, Domains I [pp. 23-46], contains just one chapter.
  - Chapter **1**, Domains I [pp. 23-46], introduces the core concept of *domains* – somewhat "novel" to most readers. It may not be suitable for primary/middle school teaching, but its "message" should be reasonably firmly embedded in the head of *Infomatics* teachers! – and the readers of this book after having studied parts **I** and **XI**.
- Part **II**, The Basics [pp. 49-153] contains 11 chapters (!). Together they provide the very foundation of *informatics*.
  - Chapter **2**, Logic [pp. 49-62], provides a simple introduction to logic (the Boolean truth values and the basic operations on these: $\sim, \vee, \wedge, =, \equiv$, etc., to be understood by all. It also sketches the bases for proofs of MoLa program and model properties – material that is somehow "more advanced".
  - Chapter **3**, Sets [pp. 63-76], likewise. Sets, as a mathematical and as a MoLa concept, are important to abstract programs and formal domain models.
  - Chapter **4**, Numbers [pp. 77-88],
  - Chapter **5**, Names and Values, Characters and Texts [pp. 89-99], has two main sections. They can be studied in any order. The first section, Sect. **5.1** is non-trivial, the second section, Sect. **5.2**, is somewhat trivial.
    - Section **5.1**, Names and Values [pp. 89-97], more to come
    - Section **5.2**, Characters and Text [pp. 97-99], is, perhaps, unusual in a book one of whose main topics is computing. But in any language, also the formal ones with which we describe domains and programs, the characters and texts, which are communicated between humans, models, programs and the computing hardware, are important.[18]
  - Chapters **6**, **8**, **10**–**11**, i.e.,

[18]**Editorial note:** Peter Naur's `ALgol 60` report [60] began with details on `Algol 60`'s characters and texts! Cf. Sects. 2.1 and 2.4! [`https://www.masswerk.at/algol60/report.htm`]

* **6**, Functions [pp. 101-105],
* **8**, Cartesians [pp. 109-117],
* **10**, Lists [pp. 135-143], and
* **11**, Maps [pp. 145-152],

constitute the main chapters on both *discrete mathematics* and MoLa "data" types. With *functions, sets, Cartesians, lists* and *maps*, MoLa "becomes" an *abstract functional program and domain model description language*. Each cover respective facets of their type: their *type expressions*, how to express *functions, Cartesians, lists* and *maps*; and *operations* on/over these.

– Chapters **7**, Infinity [pp. 107-108], and **9**, Graphs [pp. 119-133], provide "leisurely acquired" insight into the esoteric issue of *infinity* – useful in connection with the possibility of infinite sets, possibly Cartesians, and lists and maps – with Chapter **9**, Graphs [pp. 119-133], appealing to intuitive views of a variety of *graphs*.

– Chapter **12**, Types and Sorts [pp. 153-161], collects the full "story" of types, gathered from eight previous chapters, into a coherent presentation – and widens the treatment of types to that of *sorts:* abstract types, i.e., types with only implicit, indirect, type descriptions. The latter, *sorts*, become crucial for our treatment of *domain model descriptions*.

- Part **III**, Space and Time [pp. 167-182], contains three chapters:
  - Chapter **??**, textsf [pp. **??**-**??**],
  - Chapter **??**, textsf [pp. **??**-**??**],
  - Chapter **??**, textsf [pp. **??**-**??**],
- Part **IV**, Structured Clauses [pp. 185-186], contains just one chapter:
  - Chapter **??**, textsf [pp. **??**-**??**]
- Part **V**, Sequentiality and Concurrency [pp. 189-192], contains two chapters:
  - Chapter **??**, textsf [pp. **??**-**??**],
  - Chapter **??**, textsf [pp. **??**-**??**],
- Part **VI**, Algorithms [pp. 195-199], contains three chapters:
  - Chapter **??**, textsf [pp. **??**-**??**],
  - Chapter **??**, textsf [pp. **??**-**??**],
  - Chapter **??**, textsf [pp. **??**-**??**],
- Part **VII**, Programming Paradigms [pp. 203-209], contains four chapters:

- Chapter **??**, textsf [pp. **??**-**??**],
- Chapter **??**, textsf [pp. **??**-**??**],
- Chapter **??**, textsf [pp. **??**-**??**],
- Chapter **??**, textsf [pp. **??**-**??**],

• Part **VIII**, Modeling Paradigms [pp. 213-217], contains three chapters:

- Chapter **??**, textsf [pp. **??**-**??**],
- Chapter **??**, textsf [pp. **??**-**??**],
- Chapter **??**, textsf [pp. **??**-**??**],

• Part **IX**, Semiotics [pp. 221-225], contains three chapters:

- Chapter **??**, textsf [pp. **??**-**??**],
- Chapter **??**, textsf [pp. **??**-**??**],
- Chapter **??**, textsf [pp. **??**-**??**],

• Part **X**, The Triptych Dogma [pp. 229-233], contains three chapters:

- Chapter **??**, textsf [pp. **??**-**??**],
- Chapter **??**, textsf [pp. **??**-**??**],
- Chapter **??**, textsf [pp. **??**-**??**],

• Part **XI**, Domain II [pp. 237-258], contains nine chapters!

- Chapter **??**, textsf [pp. **??**-**??**],
- Chapter **??**, textsf [pp. **??**-**??**],
- Chapter **??**, textsf [pp. **??**-**??**],
- Chapter **??**, textsf [pp. **??**-**??**],
- Chapter **??**, textsf [pp. **??**-**??**],
- Chapter **??**, textsf [pp. **??**-**??**],
- Chapter **??**, textsf [pp. **??**-**??**],
- Chapter **??**, textsf [pp. **??**-**??**],
- Chapter **??**, textsf [pp. **??**-**??**],

• Part **XII**, Formal Bases [pp. 261-269], contains five chapters:

- Chapter **??**, textsf [pp. **??**-**??**],

- Chapter **??**, textsf [pp. **??**-**??**],
- Chapter **??**, textsf [pp. **??**-**??**],
- Chapter **??**, textsf [pp. **??**-**??**],
- Chapter **??**, textsf [pp. **??**-**??**],

- Part **XIII**, Closing [pp. 273-273], contains just one chapters:
  - Chapter **??**, textsf [pp. **??**-**??**],

- Part **XIV**, Appendices [pp. 285-273], contains five appendices:
  - Appendix **??**, textsf [pp. **??**-**??**],
  - Appendix **??**, textsf [pp. **??**-**??**],
  - Appendix **??**, textsf [pp. **??**-**??**],
  - Appendix **??**, textsf [pp. **??**-**??**],
  - Appendix **??**, textsf [pp. **??**-**??**],

0.14 Summary

to be written

0.15 Exercises

Exercise 1 **XIntro:**

Exercise 2 **YIntro:**

Exercise 3 **ZIntro:**

Part I

Domains I

Chapter 1

Domains, I

Contents

Motivation: Domains

The world around us is cultured by man's achievements, whether for the good or otherwise. We teach and learn physics (with chemistry), we teach and learn botanic, zoology, biology, etc. But do we actually, at school level, teach and learn about our own "creations": *the finance system, the retail industry, road, rail, ship and air transport*, etc. This book wishes to 'right' things: Let us teach, in school, how banks work, about retail supply lines, traffic control, shipping and the airline industry. For that we need to know about how to understand, i.e., analyse these kinds of main-made domains, and how they can be described. In this chapter we cover some aspects of the analysis of domains. In Part **XI** we "conclude" by covering material on their description.

> **Study Hint:** *This chapter is of central, crucial importance to the "message" of this book. It is not really intended for "direct" class teaching! Its contents should be well understood by such who teach 'Informatics'. So it should be studied by them, well, all (!). Do it at Your leisure: several more cups of tea: perhaps Indian Assam Earl Grey!*

Christians take comfort in:

> *In the beginning was the Word, and the Word was with God, and the Word was God. The same was in the beginning with God. All things were made by him; and without him was not any thing made that was made. In him was life; and the life was the light of men. And the light shineth in darkness; and the darkness comprehended it not.* [From the The Gospel of John].

We shall, perhaps profanely, interpret this *bringing together* two such diverse concepts as *"The Word"* and *the world* around us: *"All things were made by Him"*, to justify this book's emphasis on *domains* and a formal domain description *language*.

Therefore this chapter's introduction to the concept of **domains** [*All things ... made*]. And therefore most remaining chapters' introduction to a **formal domain description language**, here MoLA [*The Word*].
This paper gives an ultra-short introduction to *domain science* as developed since 2008: [17,18,20,22,23]. Since the publication of [23, *Nov. 2021*] there has been further refinements and simplifications. These are documented in the [25–29,34] reports and publications. This paper evolved as the result of writing [33, *Informatics for Beginners*]. The present paper is one of four currently being prepared: [30–32].

1.1 Domain Definition

We repeat the definition of the concept of domains as first given on Page 3.

Definition 31 Domain: *By a domain we shall understand a rationally describable segment of a discrete dynamics fragment of a human assisted reality, i.e., of the world that we daily observe. It includes its* **endurants***, i.e., solid and fluid entities of* **parts** *and* **living species***, and* **perdurants** ■

Endurants are either natural ["God-given"] or artefactual ["man-made"]. and may be considered atomic or compound parts, or, as in this book, further unanalysed living species: plants and animals – including humans.

Perdurants are here considered to be actions, events and behaviours.

We exclude, from our treatment of domains, issues of ethical, biological and psychological matters.

Example 3 Some Domain Examples: A few, more-or-less self-explanatory examples:

- **Rivers –** with their natural sources, deltas, tributaries, waterfalls, etc., and their man-made dams, harbours, locks, etc. – and their conveyage of materials (ships etc.) [24];
- **Road nets –** with street segments and intersections, traffic lights and automobiles – and the flow of these;
- **Pipelines –** with their wells, pipes, valves, pumps, forks, joins and wells and the flow of fluids [19]; and
- **Container terminals –** with their container vessels, containers, cranes, trucks, etc. – and the movement of all of these [21] ■

The definition relies on the understanding of the terms *'rationally describable'*, *'discrete dynamics'*, *'human assisted'*, *'solid'* and *'fluid'*. The last two will be explained later. By **rationally describable** we mean that what is described can be understood, including reasoned about, in a rational, that is, logical manner – in other words **logically tractable**. By **discrete dynamics** we imply that we shall basically rule out such domain phenomena which have properties which are continuous with respect to their time-wise, i.e., dynamic, behaviour. By **human-assisted** we mean that the domains – that we are interested in modelling – have, as an important property, that they possess man-made entities.

This primer presents a *method*, its *principles*, *procedures*, *techniques* and *tools*, for *analysing* &[1] *describing* domains.

1.2 A Domain Analysis & Description Ontology

Figure **1**.1 on the facing page expresses an ontology for our analysis of domains. Not an taxonomy for any one specific domain.

We refer to Fig. **1**.1 on the next page.

The idea of Fig. **1**.1 on the facing page is the following:

- It presents a recipe for how to **analyse** a domain.
- You, the *domain* **analyser** *cum describer*, are confronted[2] with, or by a domain.
- You have Fig. **1**.1 on the next page in front of you, on a piece of paper, or in Your mind, or both.

[1]We use here the ampersand, '&', as in $A\&B$, to emphasize that we are treating A and B as one concept.

[2]By 'confronted' we mean: You are reading about it, in papers, in books, in postings on the `Internet`, visiting it, talking with domain stakeholders: professional people working "in" the domain.

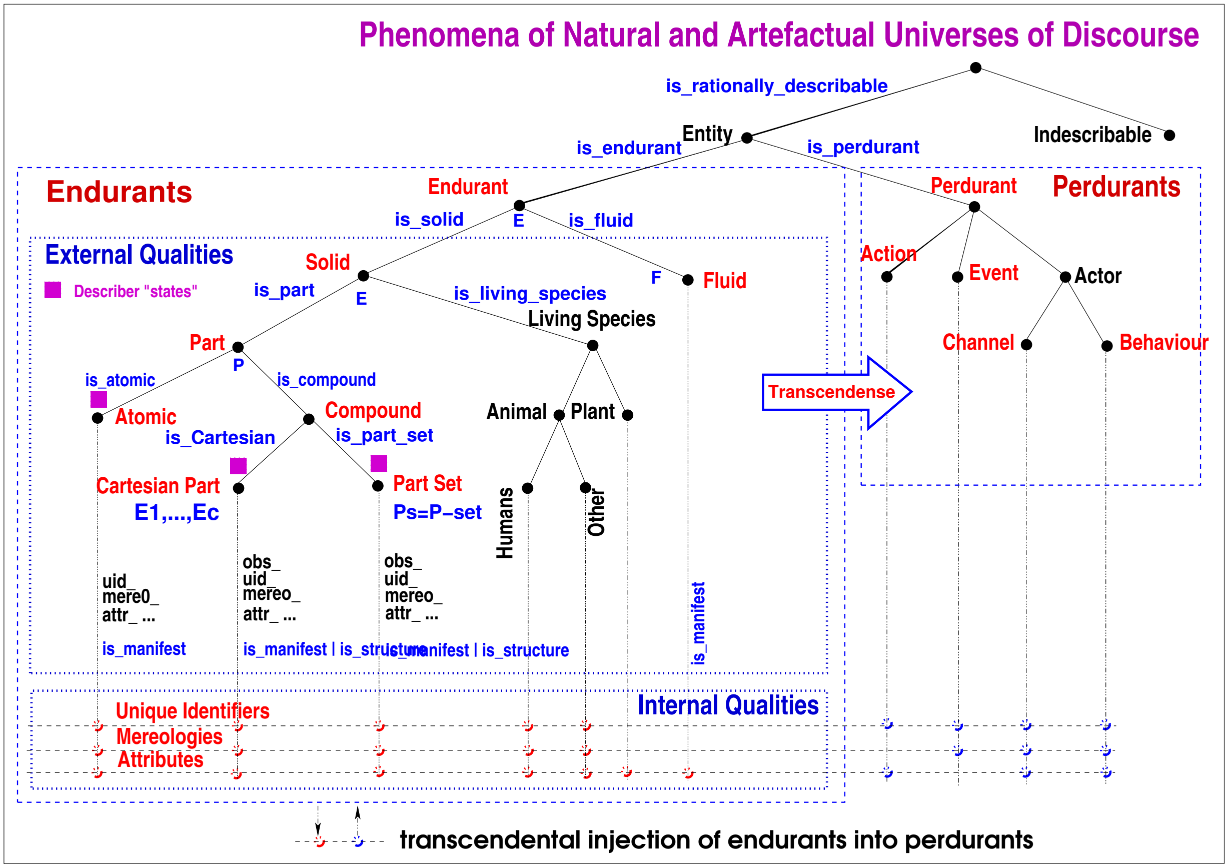


Figure 1.1: A Domain Analysis & Description Ontology

- You are then asked, by the domain **analysis** & description method of this chapter, to "start" at the uppermost •, just below and between the '**r**' and the first '**s**' in the main title, Phenomena of Natural and Artefactual Univer<u>rs</u>es of Discourse.
- The **analysis** & description ontology of Fig. **1**.1 then *directs* You to inquire as to whether the phenomenon – whichever You are "looking at/reading about/..." – is an *entity* (`is_rationally_describable`) or is *indescribable*.
- It is Your decision whether the answer to that "query" is yes or no.
- The definitions of the concepts whose names are attached to the •s of Fig. **1**.1 are given in the following sections.
- Whether they are precise enough to guide You in Your obtaining reasonable answers, "yes" or "no", to the •ed queries is, of course, a problem. I hope they are.
- If Your answer is "yes", then Your **analysis** proceeds down the tree, usually indicated by "yes" or "no" answers.

- If one, or the other is a "leaf" of the ontology tree, You have finished examining the phenomena You set out to **analyse**.
- If it is not a leaf, then further **analysis** is required.
- (We shall, in this book, leave out the analysis and hence description of *living species*.)
- If an **analysis** of a phenomenon has reached one of the (only) three ■'s, then the **analysis** at that • results in the domain describer **describing**, in MoLa, some of the properties of that phenomenon.
- That **analysis** involves "setting aside", for subsequent **analysis & description**, one or more [thus **analysis** etc.-pending] phenomena (which are subsequently to be tackled from the "root" of the ontology).

We do not [need to] prescribe in which order You analyse & describe the phenomena that has been "set aside".

1.3 The Name, Type and Value Concepts

Domain *modeling*, as well as *programming*, depends, in their *specification*, on *separation of concerns:* which kind of *values* are subjectable to which kinds of *operations*, etc., in order to achieve ease of *understanding* a model or a program, ease of *proving properties* of a model, or *correctness* of a program.

1.3.1 Names

We name things in order to refer to them in our speech, models and programs. Names of types and values in models and programs are usually not so-called "first-citizens", i.e., values that can be arguments in functions, etc. The "science of names" is interesting.[3] In `botanicalsociety.org.za/the-science-of-names-an-introduction-to-plant-taxonomy` the authors actually speak of a "science of names" in connection with plant taxonomy: the "art" of choosing such names that reflect some possible classification of what they name.

$\boxed{\text{more to come}}$

1.3.2 Types

The type concept is crucial to programming and modeling.

Definition 32 Type: *A* ***type*** *is a class of values ("of the same kind")* ■

[3]The study of names is called *onomastics* or *onomatology*. *Onomastics* covers the naming of all things, including place names (toponyms) and personal names (*anthroponyms*).

We name types.

Example 4 Type Names: Some examples of type names are:

- RT – the class of all road transport instances: the *Metropolitan London Road Transport*, the *US Federal Freeway System*, etc.
- RN – the class of all road net instances (within a road transport).
- SA – the class of all automobiles (within a road transport) ■

You, the domain describer, choose type names. Choosing type names is a "serious affair". It must be done carefully. You can choose short (as above) or long names: Road_Transport, Road_Net, etc. We prefer short, but not cryptic names, like X, Y, Z, ... Names that are easy to *memorize*.

1.3.3 Values

Values are what programming and modeling, in a sense, is all about". In programming, values are the *data* "upon" which the program code specifies computations. In modeling values are, for example, what we observe: the entities in front of our eyes.

1.4 Phenomena and Entities

Definition 33 Phenomena: *By a* phenomenon *we shall understand a fact that is observed to exist or happen* ■

Some phenomena are rationally describable – to some degree[4] – others are not.

Definition 34 Entities: *By an entity By an* entity *we shall understand a more-or-less rationally describable phenomenon* ■

Example 5 Phenomena and Entities: Some, but not necessarily all aspects of a river can be rationally described, hence can be still be considered entities. Similarly, many aspects of a road net can be rationally described, hence will be considered entities ■

[4]That is: It is up to the domain analyser cum describer to decide as to how many rationally describable phenomena to select for analysis & description. Also in this sense one practices abstraction by "abstracting away" [the analysis & description of] phenomena that are irrelevant for the "current" (!) domain description.

1.5 Endurants and Perdurants

1.5.1 Endurants

Definition 35 Endurants: *Endurants are those quantities of domains that we can observe (see and touch), in* space*, as "complete" entities at no matter which point in* time *– "material" entities that persists, endures* ■

Example 6 Endurants: Examples of endurants are: a street segment [link], a street intersection [hub], an automobile ■

1.5.2 Perdurants

Definition 36 Perdurants: *Perdurants are those quantities of domains for which only a fragment exists, in* space*, if we look at or touch them at any given snapshot in* time ■

Example 7 Perdurant: A moving automobile is an example of a perdurant ■

1.6 External and Internal Endurant Qualities

The main contribution of this section is that of a calculus of domain analysis and description prompts.

1.6.1 External Qualities

Definition 37 External Qualities: *External qualities of endurants of a manifest domain are, in a simplifying sense, those we can see, touch and have spatial extent. They, so to speak, take form.*

Example 8 External Qualities: An example of external qualities of a domains is: the Cartesian[5] of sets of solid atomic street intersections, and of sets of solid atomic street segments, and of sets of solid automobiles of a road transport system where the Cartesian, sets, atomic, and solid reflect external qualities ■

[5]Cartesian after the French philosopher, mathematician, scientist René Descartes (1596–1650)

1.6.1.1 Discrete or Solid Endurants

Definition 38 Discrete or Solid Endurants: *By a solid [or discrete] endurant we shall understand an endurant which is separate, individual or distinct in form or concept, or, rephrasing: have 'body' [or magnitude] of three-dimensions: length, breadth and depth [65, Vol. II, pg. 2046]* ■

Example 9 Solid Endurants: Examples of sold endurants are the wells, pipes, valves, pumps, forks, joins and sinks of pipelines are solids. [These units may, however, and usually will, contain fluids, e.g., oil, gas or water] ■

Type Naming: When, in a domain analysis, we encounter a solid, for the first time, we name its type, i.e., anticipating the upcoming solid description, as for parts, i.e., atomic, compound, Cartesian and part set parts (or for living species)[6], see below, we "set aside", somehow, say in our mind, or on a piece of paper, or in a computer document, that or those type names.

1.6.1.2 Fluids

Definition 39 Fluid Endurants: *By a fluid endurant we shall understand an endurant which is prolonged, without interruption, in an unbroken series or pattern; or, rephrasing: a substance (liquid, gas or plasma) having the property of flowing, consisting of particles that move among themselves [65, Vol. I, pg. 774]* ■

Example 10 Fluid Endurants: Examples of fluid endurants are: water, oil, gas, compressed air, smoke ■

Fluids are otherwise liquid, or gaseous, or plasmatic, or granular[7], or plant products, i.e., chopped sugar cane, threshed, or otherwise[8], et cetera. Fluid endurants will be analysed and described in relation to solid endurants, viz. their "containers".

Type Naming: When, in a domain analysis, we encounter a fluid, for the first time, we name its type, i.e., anticipating the upcoming fluid description, we "set aside", somehow, say in our mind, or on a piece of paper, or in a computer document, that or those type names.

1.6.1.3 Parts

Definition 40 Parts: *The non-living species solids are what we shall call parts* ■

---

[6]– whose further analysis we shall not cover in this book

[7] This is a purely pragmatic decision. "Of course" sand, gravel, soil, etc., are not fluids, but for our modelling purposes it is convenient to "compartmentalise" them as fluids!

[8]See footnote 7.

Parts are the "work-horses" of man-made domains. That is, we shall mostly be concerned with the analysis and description of endurants into parts.

Example 11 **Parts:** The previous example of solids was also an example of parts ■

We distinguish between atomic and compound parts.

1.6.1.3.1 Atomic Parts.

Definition 41 **Atomic Part, I:** *By an* atomic part *we shall understand a part which the domain analyser considers to be indivisible in the sense of not meaningfully consist of sub-parts* ■

Example 12 **Atomic Parts:** Examples of atomic parts are: hubs, i.e., street intersections; links, i.e., the stretches of roads between two neighbouring hubs; and automobiles:

type H, L, A ■

1.6.1.3.2 Compound Parts.

We, pragmatically, distinguish between Cartesian product- and set-oriented parts.

Definition 42 **Compound Part, I:** *Compound parts are those which are observed to [potentially] consist of several parts* ■

Example 13 **Compound Parts:** An example of a compound parts is: a road net consisting of a set of hubs, i.e., street intersections or "end-of-streets", and a set of links, i.e., street segments (with no contained hubs), is a Cartesian compound; and the sets of hubs and the sets of links are part set compounds ■

1.6.1.3.3 Cartesians.

Definition 43 **Cartesians:** *Cartesian parts are those compound parts which are observed to consist of two or more distinctly sort-named endurants (solids or fluids)* ■

Example 14 **Cartesians: Road Transport:** A road transport, rt:RT, is observed to consist of an aggregate of a road net, rn:RN, and a set of automobiles, SA, where the road net is observed, i.e., abstracted, as a Cartesian of a set of hubs, ah:AH, i.e., street intersections (or specifically designated points segmenting an otherwise "straight" street into two such), and a set of links, al:AL, i.e., street segments between two "neighbouring" hubs.

type
  RT, RN, SA, AH = H-**set**, AL = L-**set**
value
  obs_RN: RT → RN, **obs**_SA: RT → SA,, **obs**_AH: RN → AH, **obs**_AL: RN → AL ■

Once a part has been analysed into a Cartesian, say p:P, we inquire as to the type names of the endurants[9] of which it consists. The inquiry: **record_Cartesian_parts**(p:P), we decide, then yields the type of the constituent endurants.

1: record_ Cartesian_part_type_names(p:P)

value
  record_Cartesian_part_type_names: P → $\mathbb{T}$-**set**
  record_Cartesian_part_type_names(p) **as** {ηE1,ηE2,...,ηEn}

Here $\mathbb{T}$ is the **name** of the type of all type names, and η Ei is the **name** of type Ei.

Example 15 **Cartesian Parts:** The *Cartesian part*s of a road transport, rt:RT, is thus observed to consists of

- an aggregate of a road net, rn:RN, and
- an aggregate set of automobiles, sa:SA:

that is:

- **record_Cartesian_part_type_names**(rt:RT) = {η RN,η SA}

where the type names η RT were and η RN and η SA are coined, i.e., more-or-less freely chosen, by the domain analyzer cum describer ∎

1.6.1.3.4 Part Sets.

Definition 44 **Part Sets:** *Part sets are those compound parts which are observed to consist of an indefinite number of zero, one or more parts* ∎

Once a part has been analysed into a part set, say s:S, we inquire as to the set of parts and their type of which it consists. The inquiry: **record_part_set_parts**(s:S), we decide, then yields the (single) type of the constituent parts.

2: record_ part_set_part_type_names(s:S)

value
  record_part_set_part_type_names: S → $\mathbb{T}$Ps×$\mathbb{T}$P
  record_part_set_part_type_names(s:S) **as** (η Ps,η P)

Here the name of the value, s, and the type names η S and η P are coined, i.e., more-or-less freely chosen, by the domain analyzer cum describer ∎

[9]We emphasize that the observed elements of a Cartesian part may be both solids, at least one, and fluids

Example 16 Part Sets: Road Transport: The road transport contains a set of automobiles. The part set type name has been chosen to be SA. It is then determined (i.e., analyzed) that SA is a set of Automobile of type A

- **record_part_set_parts**(sa:SA) = ηA

• • •

So far we have only touched upon the 'External Qualities' labeled, dotted-dashed box of the 'Endurants'-labeled dashed box of Fig. **1**.1.

1.6.1.4 Compound Observers

Once the domain analyser cum describer has decided upon the names of atomic and compound parts, **obs**_erver functions can be applied to Cartesian, c:C, respectively part set, ps:PS, parts:

3: describe_ Cartesians and Part Set Parts

```
value
    let {ηP1,ηP2,...,ηPn} = record_Cartesian_part_type_names(c:C) in
    "type
            P1, P2, ..., Pn;
      value
            obs_P1: C→P1, obs_P2: C→P2,...n obs_Pn: C→Pn "
                                                            [respectively:]
    let {ηPs,ηP} = record_part_set_part_type_names(ps:PS) in
    "type
              Ps = P-set,
      value
              obs_Ps: C→Ps "
    end end
```

The **"..."** texts are the MoLa texts "generated", i.e., written down, by the domain describer. They are *domain model specification units*.

The "surrounding" MoLa-like texts are not written down as phrases, elements, of the domain description. They are elements of the domain describers' "notice board", and, as such, elements of the development of domain models.

We have thus introduced a core domain modeling tool the **obs_**... observer function, one to be "applied" mentally by the domain describer, and one that appears in (MoLa) domain descriptions

The **obs_**... observer function is "applied" by the domain describer, it is not a computable function.

1.6.1.5 States

Definition 45 **States:** *By a* state *we shall mean any subset of the parts of a domain* ■

Example 17 **Road Transport State:**

variable
hs:AH := **obs_**AH({\obs}RN(rt)),
ls:AL := **obs_**AL({\obs}RN(rt)),
as:SA := **obs_**SA(rt),
σ:(H|L|A)-**set** := $hs \cup ls \cup as$ ■

We have chosen to model domain states as **variable**s rather than as **value**s. The reason for this is that monitorable, including biddable part attributes change, and that domains are often extended and "shrunk" by the addition, respectively removal of parts: **Example 18 Road Transport Development:** adding or removing hubs, links and automobiles in a road transport ■ We do not cover the aspect of bidding changes to monitorable part attributes, nor the introduction of new parts and removal of former parts in this paper.

1.6.1.6 Validity of Endurant Observations

We remind the reader that the **obs_**erver functions, as all later such functions: **uid_**-, **mereo_**- and **attr_**-functions, are applied by humans and that the outcome of these "applications" is the result of human choices, and possibly biased by inexperience, taste, preference, bias, etc.

How do we know whether a domain analyser & describer's description of domain parts is valid ? Whether relevantly identified parts are modeled reasonably wrt. being atomic, Cartesians or part sets Whether all relevant endurants have been identified ? Etc. The short answer is: we never know. Our models are conjectures and may be refuted[10]. A social process of peer reviews, by domain stakeholders and other domain modelers is needed.

1.6.1.7 Summary of Analysis Predicates

- **Endurant Ontology:**
  - is_entity
  - is_endurant
  - is_perdurant
  - is_solid
  - is_fluid
  - is_part
  - is_living_species
  - is_atomic
  - is_compound
  - is_Cartesian
  - is_part_set
  - is_plant
  - is_animal
  - is_human
- **Location:**
  - is_stationary
  - is_mobile
- **Treatment:**
  - is_manifest
  - is_structure

[10] We refer to [71, *Sir Karl Popper*].

Example 19 Road Transport: Automobiles are **mobile**, hubs and links are **stationary**, only Automobiles, hubs and links are **manifest**, the rest are **structures**, i.e., conceptual ■

1.6.2 Internal Qualities

Definition 46 Internal Qualities: *Internal qualities are those properties [of endurants] that do not occupy space but can be measured or spoken about* ■

Example 20 Internal qualities: Examples of internal qualities are the *unique identity* of a part, the *mereological relation* of parts to other parts, and the endurant *attributes* such as temperature, length, colour ■

This section therefore introduces a number of domain description tools:

- **uid_**: the unique identifier observer of parts;
- **mereo_**: the mereology observer of parts;
- **attr_**: (zero,) one or more attribute observers of endurants; and
- **attributes_**: the attribute query of endurants.

1.6.2.1 Unique Identity

Definition 47 Unique Identity: *A unique identity is unique identity an immaterial property that distinguishes any two spatially distinct solids* ■

The unique identity of a part p of type P is obtained by observer **uid_**P:

4: describe_ Unique Identity Part Observer

```
type
    P,PI
value
    uid_P: P → PI
```

Here PI is the type of the unique identifiers of parts of type P.

Example 21 Unique Identifiers: The *unique identifiers*s of a road transport, rt:RT, is thus observed, under **record_unique_identifiers(rts:RTS)**, to consists of the unique identifiers of the

- road transport – rtsi:RTSI,
- (Cartesian) road net – rni:RNI,
- (set of) automobiles – sa:SAI,
- automobile, ai:AI,

- (set of) hubs, hai:AHI,
- (set of) links, lai:LAI,
- hub, hi:HI, and
- link, li:LI,

that is:

- **record_unique_identifiers**(rt:RT) = {RNI,SAI,AI,AHI,LAI,HI,LI}

where the type names are all coined, i.e., more-or-less freely chosen, by the domain analyzer cum describer – though, as You can see, these names were here formed by "suffixing" Is to relevant part names ■

1.6.2.1.1 Unique Identity Observer Functions. Once a domain has been analysed into all its parts, we can ascertain these by an informal function. The inquiry **domain_part_type_names**(rts:RTS)[11] yields the type names of their unique identifiers – and hence of their observer functions, **uid_**P[12]:

5: describe_ unique_identifiers

let {η P1,η P2,...,η Pn} = **record_domain_part_type_names**(rts:RTS) **in**
"**type**
P1I, P2I, ..., PnI;
value
uid_P1: P1→P1I, **uid_**P2: P2→P2I,..., **uid_**Pn: Pn→PnI "
end

The **"..."** texts are the MoLA texts "generated", i.e., written down, by the domain describer. They are *domain model specification units*.

We have thus introduced a core domain modeling tool the **uid_**... observer function, one to be "applied" mentally by the domain describer, and one that appears in (MoLA) domain descriptions

The **uid_**... observer function is "applied" by the domain describer, it is not a computable function.

1.6.2.1.2 Uniqueness of Parts No two parts have the same unique identifier.

Example 22 **Road Transport Uniqueness:**

variable
hs_{uids}:AHI-**set** := { **uid_**(h) | h:H•u ∈ σ }
ls_{uids}:ALI-**set** := { **uid_**(l) | l:L•u ∈ σ }

---

[11] We leave it to the reader to define **domain_part_type_names**(rts:RTS).
[12] Cf. Sects. **1.6.1.3.3**– **1.6.1.4** on page 34

as_{uids}:SAI-**set** := { **uid_**(a) | a:A•u ∈ σ }
σ_{uids}:(HI|LI|AI)-**set** := { **uid_**(u) | u:(H|L|A)•u ∈ σ }
axiom
□ **card** σ = **card** σ_{uids} ∎

We have chosen, for the same reason as given in Sect. **1.6.1.5** on page 35, to model a unique identifier state. The □ prefix in the **axiom** then expresses that changes of parts or addition of parts to and deletions of parts from the domain shall maintain their uniqueness.

1.6.2.2 Mereology

Definition 48 **Mereology, I:** *Mereology is a theory of [endurant] part-hood relations: of the relations of an [endurant] parts to a whole and the relations of [endurant] parts to [endurant] parts within that whole* ∎

Example 23 **Mereology:** Examples of mereologies are that a link is topologically *connected* to exactly two specific hubs, that hubs are *connected* to zero, one or more specific links, and that links and hubs are *open* to specific subsets of automobiles ∎

Mereologies can be expressed in terms of unique identifiers.

Example 24 **Mereology Representation:** For our 'running road transport example' the mereologies of links, hubs and automobiles can thus be expressed as follows:

- **mereo_**L(l) = {hi′,hi″} where hi,hi′,hi″ are the unique identifiers of the hubs that the link connects, i.e., are in hs_{uids};
- **mereo_**H(h) = {li$_1$,li$_2$,...,li$_n$} where li$_1$,li$_2$,...,li$_n$ are the unique identifiers of the links that are imminent upon (i.e., emanates from) the hub, i.e., are in ls_{uids}; and
- **mereo_**A(a) = {ri$_1$,ri$_2$,...,ri$_m$} where ri$_1$,ri$_2$,...,ri$_m$ are unique identifiers of the road (hub and link) elements that make up the road net, i.e., are in $hs_{uids} \cup ls_{uids}$ ∎

Once the unique identifiers of all parts of a domain has been described we can analyses and describe their mereologies. The inquiry: **mereo_P**(p) yields a mereology type (name), say PMer, and its description[13]:

6: describe_ Mereology

"**type**
  PM = MereoP
value
  mereo_P: P → PM

[13] Cf. Sects. **??**– **1.6.1.4** on page 34

axiom
$\mathcal{A}$(pm:PM)"

where MereoP is a type expression over unique identifier types of the domain; **mereo**_P is the mereology observer function for parts p:P; and $\mathcal{A}$(pm:P) is an axiom that secures that the unique identifiers of any part are indeed of parts of the domain.

1.6.2.3 Attributes

1.6.2.3.1 General.

Definition 49 Attributes: *Attributes are properties of endurants that are not spatially observable, but can be either physically (electronically, chemically, or otherwise) measured or can be objectively spoken about* ■

Attributes are of types and, accordingly have values.

7: record_ attribute_type_names

- **value**

  record_attribute_type_names: $\mathrm{P} \to \eta\mathbb{T}$**-set**

  record_attribute_type_names(p:P) as $\eta\mathrm{T}$**-set**

Example 25 Road Net Attributes, I: Examples of attributes are: hubs have states, hσ:HΣ: the set of pairs of link identifiers, ($f$li,$t$li), of the links $f$rom and to which automobiles may enter, respectively leave the hub, hubs have state spaces, hω:HΩ: the set of hub states "signaling" which states are open, i.e., **green**; links that have lengths, LEN; and automobiles have road net positions, APos, either *at a hub*, atH, or *on a link*, onL, some fraction, f:**Real**, down a link, identified by li, from a hub, identified by fhi, towards a hub, identified by thi.

type
  HΣ = (LI×LI)-**set**
  HΩ = HΣ-**set**
  LEN = **Nat m**
  APos = atH | onL
  atH :: HI
  onL :: LI × (fhi:HI × f:**Real** × thi:HI)
value
  attr_HΣ: H → HΣ
  attr_HΩ: H → HΩ
  attr_LEN: L → LEN
  attr_APos: A → APos
axiom
  ∀ (li,(fhi,f,thi)):onL • 0<f<1
    ∧li∈ls_{uids}∧{fhi,thi}⊆hs_{uids}∧... ■

8: describe_ Endurant attributes of e:E

```
let {η A1,ηA2,...,ηAn}  = record_attribute_type_names(e:E) in
"type
      A1, A2, ..., An
   value
      attr_A1: E → A1, attr_A2: E → A2, ..., attr_An: E → An
   axiom
      ∀ a1:A1, a2:A2, ..., an:An: 𝒜(a1,a2,...,an) "
end
```

1.6.2.3.2 Michael A. Jackson's Attribute Categories. Michael A. Jackson [58] has suggested a hierarchy of attribute categories: from *static* to *dynamic* values – and within the dynamic value category: *inert* values, *reactive* values, *active* values – and within the dynamic active value category: *autonomous* values, *biddable* values and *programmable* values. We refer to [59, *M. A. Jackson*] and [23, *Chapter 5, Sect. 5.4.2.3*] for details. We summarize Jackson's attribute categorization in Fig. **1**.2.

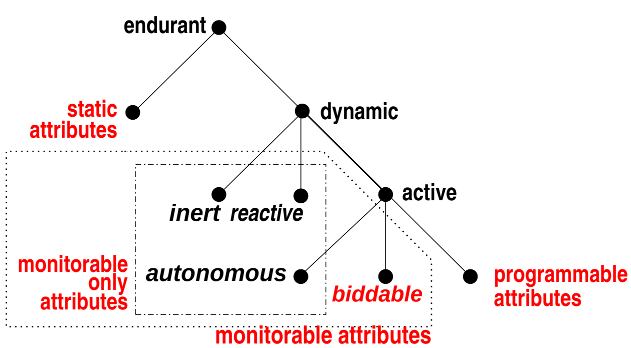


Figure 1.2: Michael Jackson's Attribute Categories

Example 26 Road Net Attributes, II: The link length and hub state space attributes are static, hub states and automobile positions programmable. Automobile speed and acceleration attributes, which we do not model, are monitorable ■

The attributes categorization determines, in the next major section on perdurants, the treatment of hub, link and automobile behaviours.

1.6.3 Intentional Pull

to be written

1.7 Perdurant Concepts

The main contribution of this section is that ***transcendentally deducing*** perdurants from endurant parts, in particular ***behaviours*** "of" parts.

1.7.1 "Morphing" Parts into Behaviours

As already indicated we shall transcendentally deduce (perdurant) behaviours from those (endurant) parts which we, as domain analysers cum describers, have endowed with all three kinds of internal qualities: unique identifiers, mereologies and attributes. We shall use the `CSP` [55] constructs of `RSL` [47] to model concurrent behaviours.

1.7.2 Actors – A Synopsis

This section provides a summary overview.

Definition 50 **Actors:** *An actor is anything that can initiate an* **action**, *an* **event** *or a* **behaviour** ■

1.7.2.1 Action

Definition 51 **Actions:** *An action is a function that can purposefully change a state* ■

Example 27 **Road Net Actions:** These are some road transport actions: an automobile leaving a hub, entering a link; leaving a link, entering a hubs; entering the road net; and leaving the road net ■

1.7.2.2 Event

Definition 52 **Events:** *An event is a function that surreptitiously changes a state* ■

Example 28 **Road Net Events:** These are some road net events: The blocking of a link due to a mud slide; the failing of a hub traffic signal due to power outage; an automobile failing to drive; and the blocking of a link due to an automobile accident ■

We shall not formalize events.

1.7.2.3 Behaviour

Definition 53 Behaviours: *A behaviour is a set of sequences of actions, events and behaviours* ■

Concurrency is modeled by the ***sets*** of sequences. Synchronization and communication of behaviours are effected by CSP ***output/inputs***: ch[{i,j}] !value/ch[{i,j}] ?.

Example 29 Road Net Traffic: Road net traffic can be seen as a behaviour of all the behaviours of automobiles, where each automobile behaviour is seen as sequence of start, stop, turn right, turn left, etc., actions; of all the behaviours of links where each link behaviour is seen as a set of sequences (i.e., behaviours) of "following" the link entering, link leaving, and movement of automobiles on the link; of all the behaviours of hubs (etc.); of the behaviour of the aggregate of roads, viz. *The Department of Roads*, and of the behaviour of the aggregate of automobiles, viz, *The Department of Vehicles*.

We shall, in this paper, consider aggregates as purely conceptual structures. That is, structures have no behaviour (counterparts).

1.7.3 Channel

Definition 54 Channel: *A channel is anything that allows synchronisation and communication of values between two behaviours* ■

Example 30 Road Transport Interaction Channel:

channel { ch[{ui,uj}] | {ui,ij}:(HI|LI|AI)-**set** • ui≠uj∧{ui,uj}$\subseteq\sigma_{uids}$ } M

Channel array ch is indexed by a "pair" of distinct unique hub, link and automobile identifiers of the domain. We shall later outline M, the type of the "messages" communicated between automobile, hub and link behaviours ■

1.7.4 Behaviours

We single out the perdurants of behaviours – as they relate directly to the parts of Sect. **1.6**. The treatment is "divided" into three sections.

1.7.4.1 Behaviour Signature

Definition 55 Behaviour Signature: *By the* ***behaviour signature****, for a part p, we shall understand a pair: the name of the behaviour and a function type expression as indicated:*

value

p: $\mathsf{Uid}_p\rightarrow$[14] $\mathsf{Mereo}_p\rightarrow\mathsf{Sta\_Vals}_p\rightarrow\mathsf{Mon\_Refs}_p\rightarrow\mathsf{Prgr\_Vals}_p$ → { ch[{i,j}] | ... } **Unit**

[14] We have Schönfinckel'ed `https://en.wikipedia.org/wiki/Moses_Schönfinkel#Further_reading` (Curried `https://en.wikipedia.org/wiki/Currying`) the function type

where Uid_p is the type of unique identifiers of part p; Mereo_p is the type of the mereology of part p; $\mathsf{Sta\_Vals}_p$ is a Cartesian[15] of the type of static attributes of part p; $\mathsf{Mon\_Refs}_p$ is a Cartesian[16] of the names of the types of monitorable attributes of part p; $\mathsf{Prgr\_Vals}_p$ is a Cartesian[17] of the type of programmable attributes of part p; { ch[{i,j}] | ... } specifies the channels over which part p may communicate; and **Unit** is the type name for the () value[18] ■

Example 31 Road Transport Behaviour Signatures:

value

hub: HI→MereoH→(HΩ×...)→(...)→(HHist×...)
    →{ch[{**uid**_H(p),ai}]|ai:AI•ai∈as_{uid}} **Unit**
link: LI→MereoL→(LEN×...)→(...)→(LHist×...)
    →{ch[{**uid**_L(p),ai}]|ai:AI•ai∈as_{uid}} **Unit**
automobile: AI→MereoA→(...)→(ηAVel×ηHAcc×...)→(APos×AHist×...)
    →{ch[{**uid**_H(p),ri}]|ri:(HI|LI)•ri∈hs_{uid}∪ls_{uid}} **Unit**

Here we have suggested additional part attributes: programmable hub and link histories, HHist, LHist, monitorable automobile velocity and acceleration, AVel, AAcc; and omitted other such part attributes: '...' ■

1.7.4.2 Behaviour Description

Behaviour descriptions rely strongly on CSPs' [55] expressivity. Leaving out some details (_, '...'), and *without "further ado"*, we exemplify.

Example 32 Automobile Behaviour at Hub:

1. We abstract automobile behaviour at a Hub (hi).
   (a) Either the automobile remains in the hub,
   (b) or, internally non-deterministically,
   (c) leaves the hub entering a link,
   (d) or, internally non-deterministically,
   (e) stops.

```
1   automobile(ai)(ris)(...)(apos:atH(hi),_) ≡
1a      automobile_remains_in_hub(ai)(ris)(...)(apos:atH(hi),_)
1b     ⌈⌉
1c      automobile_leaving_hub(ai)(ris)(...)(apos:atH(hi),_)
1d     ⌈⌉
1e      automobile_stop(ai)(ris)(...)(apos:atH(hi),_)
```

[15]– where the Cartesian may "degenerate" to the non-Cartesian of no, or just one type identifier
[16]cf. footnote 15
[17]cf. footnote 15
[18]– You may "read' () as the value yielded by a never-terminating function

2. [1a] The automobile remains in the hub:
    (a) time is recorded,
    (b) the automobile remains at that hub, "idling",
    (c) informing ("first") the hub behaviour.

```
2  automobile_remains_in_hub(ai)(ris)(...)(apos:atH(hi),_) ≡
2a     let τ = record_𝕋𝕀𝕄𝔼 in
2c     ch[ {ai,hi} ] ! τ ;
2b     automobile(ai)(ris)(...)(apos:atH(hi),_)
2    end
```

3. [1c] The automobile leaves the hub entering link li:
    (a) time is recorded;
    (b) hub is informed of automobile leaving and link that it is entering;
    (c) "whereupon" the vehicle resumes (i.e., "while at the same time" resuming) the vehicle behaviour positioned at the very beginning (0) of that link.

```
3  automobile_leaving_hub(ai)({li}∪ris)(...)(apos:atH(hi),_) ≡
3a      let τ = record_𝕋𝕀𝕄𝔼 in
3b      (ch[ {ai,hi} ] ! τ ∥ ch[ {ai,li} ] ! τ) ;
3c      automobile(ai)(ris)(...)(onL(li,(hi,0,_))) end
3    pre: [hub is not isolated]
```

The choice of link entered is here expressed (3) as a non-deterministic choice[19]. One can model the leave hub/enter link otherwise.

4. [1e] Or the automobile "disappears — off the radar" !

```
4  automobile_stop(ai)(ris),(...)(apos:atH(hi),_) ≡ stop ■
```

[19]– as indicated by the **pre**- condition: the hub mereology must specify that it is not isolated. Automobiles can never leave isolated hubs.

1.7.4.3 Behaviour Initialization

For every manifest part it must be described how its behaviour is initialized.

Example 33 Road Transport Initialization: We "wrap up" the main example of this paper: We omit treatment of monitorable attributes.

5. Let us refer to the system initialization as an action.
6. All hubs are initialized,
7. all links are initialized, and
8. all automobiles are initialized.

value
5. rts_initialization: **Unit** $\rightarrow$ **Unit**
5. rts_initialization() $\equiv$
6. $\|$ { hub(**uid**_H(l))(**mereo**_H(l))(**attr**_HΩ(l))(**attr**_HΣ(l))| h:H • h $\in$ hs }
7. $\|$ $\|$ { link(**uid**_L(l))(**mereo**_L(l))(**attr**_LEN(l),...)(**attr**_LΣ(l))| l:L • l $\in$ ls }
8. $\|$ $\|$ { automobile(**uid**_A(a))(**mereo**_A(a))(**attr**_APos(a)) | a:A • a $\in$ as }

We have here omitted possible monitorable attributes. For hs, ls, as we refer to Sect. **1.6.1.5** on page 35 ■

1.8 Closing

to be written

1.8.1 Summary

This chapter has introduced the main concepts of domains such as we shall treat (analyse and describe) domains.[20] The next many chapters shall now systematically treat the domain description language, MoLa, and the analysis and description of domains, including the domain of computing! That treatment takes concept by concept and provides proper definitions and introduces appropriate analysis and description prompts; one-by-one, in an almost pedantic, hence perhaps "slow" progression!

In-between, here-and-there, we shall "smuggle" concepts of *domain analysis & description* tools and techniques. These are then covered more systematically in Chapter **32**.

1.8.2 Conclusion

to come

[20]We have omitted treatment of *living species: plants* and *animals* – the latter including *humans*.

1.9 Exercises

Exercise 4 XDom:

Exercise 5 YDom:

Exercise 6 ZDom:

Part II

The Basics

Chapter 2

Logic

Contents

In this chapter we shall introduce the programming cum modelling language MoLa's concept of **Booleans**. George Boole (1815–1864) was an English mathematician, philosopher, and logician. Boolean logic is credited with laying the foundations for the Information Age.

Motivation: Logic

Logic is an indispensable tool in analyzing and understanding domains.

Study Hint: *This chapter covers two "faces" of logic. The first "face", Sects.* **2.1**–**2.6**, *cover essential, basic material while Sect.* **2.7** *covers more advanced material. Students should learn the basics, teachers the advance material.*

To **reason**[1] is to form **logic judgments**[2] – to express **assertions**[3]. **Logic** is the **study** and **practice** of **correct**[4] **reasoning**. In all of mathematics and in all of the exact sciences their practitioners use logic to argue and to conclude their findings – *"Logic has the important function of saying what follows from what"* [62, Chapter 1 § 1].

2.1 A Basis for Logic

Based on the Danish philosopher Kai Sørlander's work [83–88] we postulate the following.

---

[1]**reason:** the power of the mind to think, understand, and form judgments logically
[2]**judgment:** the ability to make considered decisions or come to sensible conclusions
[3]**assertion:** a confident and forceful statement of fact or belief
[4]**correct:** the quality or state of being free from error, accurate

2.1.1 The Possibility of Truth

The **possibility of truth**, that an *assertion* – made by humans – may either hold, be true, or not hold, i.e., be false, is shared by us all. That is, the *possibility of truth*, is taken for granted.

2.1.2 The Principle of Contradiction

Once we accept that *the possibility of truth* cannot be denied, we have also accepted **the principle of contradiction**, that is, that an assertion and its negation cannot both be true.

2.2 Truth Values true, false and Type Bool

2.2.1 Truth Values

So how do we express truth values. Well, no one has ever seen the truth values. But we can give them names:

- **true, false**

true and **false** are literals. They are part of MoLa.

2.2.2 Truth Type

By a *type* we mean a class of values. We can give names to types. The name of the truth type is **Bool**.[5] **Bool** is a literal. It is part of MoLa.

2.3 Motivation: Simple Reasonings

To see, better, why logic "surrounds us", i.e., is an indispensable element in our daily life: reflections and communications, we present some informal examples.

Example 34 Informal Correct Reasoning, I:

9. Two plus three equals five.

10. Two plus three is not equal to six ■

[5] **Bool**, for Boolean, in honour of the mathematician George Boole, 1815–1864, professor at Queen's College, Cork in Ireland.

In the *"Two plus three equals five"* example there are three **sub-terms:** (a) *"Two plus three"*, (b) *"equals"* and (c) *"five"*. Terms (a) and (c) stand for **numbers**, in this case 2, respectively 3. Term (b) stand for an equality **relation** between [whatever] (a) and (c) stands for. That relation either **holds**, or does **not hold**.

The relation in the *"Two plus three is not equal to six"* is *"not equal"*; it is the **negation** of an equality relation. That relation either **holds**, or does **not hold**.

Exercise 7 Correct Reasoning: *The teacher asks the students, in groups of 1 or 2, to work out a number of examples of correct reasoning. These exercises are then to be discussed in class, as lead by the teacher, as to their meaningfulness* ■

Example 35 Informal Incorrect Reasoning: Two examples of syllogisms[6]:

11. The Eiffel Tower is higher than the sounds of the bells of the Notre Dame church.
12. A stone is deaf. My grandmother is deaf. So my grandmother is a stone ■

In the *"The Eiffel Tower is higher than the sounds of the bells of the Notre Dame church"* example the **truth relation:** *"is higher"* **fails**. It fails because its to operands: *"The Eiffel Tower"* and *"the sounds of the bells of the Notre Dame church"* are **in-commensurable:** cannot be **compared**. We say that these to terms, the operands, are of different **type**. One is of type, say building, with **attribute** height, in this case a little over 300 meters; the other is of type bell, with **attribute** decibel (dB, power of sound), in some cases 85dB.

The *type* concept, basically absent from mathematics, is of overwhelming importance to us: in informatics and computer science. We shall use to the concept of type in all chapters and focus on it, in particular, in Sect. **6.1** and in Chapter **12**.

Exercise 8 Incorrect Reasoning: *The teacher asks the students, in groups of 1 or 2, to work out a number of examples of incorrect reasoning. These exercises are then to be discussed in class, as lead by the teacher, as to their incorrectness* ■

2.4 Logical Operations and Relations

2.4.1 General Form of Assertions

The general forms of logical assertions are

- *it is the case that "such-and-such" is* **true**.
- *it is* **not** *the case that "such-and-such" is* **true**.

If we replace the **not** by $\sim$ and the *"such-and-such"* with b and otherwise abbreviate, then the above can be expressed as:

- b respectively $\sim b$

[6]**syllogism:** an instance of a form of reasoning in which a conclusion is drawn from two given or assumed propositions (premises); a common or middle term is present in the two premises but not in the conclusion, which may be invalid (e.g. *all dogs are animals; all animals have four legs; therefore all dogs have four legs*).

2.4.2 Informal and "Formal" Assertions

In example 34 on page 51 there were two informal examples:

13. "*Two plus three* **equal***s [=] five*" and
14. "*Two plus three is* **not equal** *[≠] to six*".

We illustrate, now, how they can be "formalised":

13. $(2{+}3) = 5$
14. $(2{+}3) \neq 5$

We can extend example 34 on page 51 by yet some informal examples:

Example 36 Informal Correct Reasoning, II:

15. "*Two plus three* **equals** *[=] five* **and** *[∧] two plus three* **is not equal** *[≠] to six*":
    $(((2+3) = 5)) \wedge ((2+3){\neq}6)$,
16. "*Two plus three* **equals** *five* **or** *[∨] two plus three* **is equal** *to six*":
    $(((2+3) = 5) \vee (2+3)){=}6$,
17. "*Two plus three* **equals** *five* **is not** *[≠]* **false**":
    $\neg(\neg((2+3) = 5))$

Etcetera:

18. "*Two plus three is not equal to six*":
    $(2+3) \neq 6$
19. "*Two plus three equals five and two plus three is not equal to six*"
20. "*Two plus three equals five or two plus three is equal to six*"
21. "*Two plus three equals five is not* **false**"
22. "*Two plus three is not equal to six*"
23. "*Two smaller than three and three smaller than four imply two plus three smaller than seven*" and
24. "*It is not the case that the Eiffel Tower is higher than the sounds of the bells of the Notre Dame church*" ∎

2.5 Model Theory

In mathematical logic, model theory is the study of the relationship between formal theories (a collection of sentences in a formal language expressing statements about a mathematical structure), and their models (those structures in which the statements of the theory hold)[7].

The formal language statements are here expressed in terms of the Boolean connectives: $\wedge, \vee, \sim, \supset$, and $\equiv$; and the models are here expressed in terms of the truth tables of this section.

The examples of the previous section illustrate combinations of assertions by means of the logic[al] operators:

- *"and"*, $\wedge$,
- *"or"*, $\vee$,
- *"not"* $\sim$, and
- *"imply"*, $\supset$,

and the relation[al] operator:

- *"is equivalent to"* $\equiv$.

We shall now explain the operations denoted by these operators and the *"is equivalent to"* relation. We shall assume that the arithmetic operators *"plus"* etc. $+, -, *, /$ and arithmetic, i.e., number, relations *"smaller"* etc. $<, \leq, =, \neq, >, \geq$ are known to the reader.

2.5.1 Negation: "not", $\sim$

The general form of a negated logical assertion was $\sim b$.

- *it is* **not** *the case that "such-and-such" is* **true**.

We explain the "meaning" of **not**, i.e., $\sim$, as it appears in $\sim b$:

- If b is **true** then $\sim b$ is **false**.
- If b is **false** then $\sim b$ is **true**.
- If b is **chaos** then $\sim b$ is **chaos**.

We need this last clause to explain the meaning of Example 36.20 on the preceding page. This is summarised in Table **2**.1 on the next page.

2.5.2 Conjunction: "and", $\wedge$

We proceed to explain the conjunction operator as it appears in $a \wedge b$.

- If a is **true** and b is **true** then $a \wedge b$ is **true**.
- If a is **true** and b is **false** then $a \wedge b$ is **false**.

---

[7] Chang and Keisler: https://books.google.dk/books?id=uiHq0EmaFp0C&pg=PA1&redir_esc=y#v=-onepage&q&f=false

$\sim$	
true	**false**
false	**true**
chaos	**chaos**

Table **2**.1: Negation: *"not"*, $\sim$

- If a is **true** and b is **chaos** then $a \wedge b$ is **chaos**.
- If a is **false** and b is **true** then $a \wedge b$ is **false**.
- If a is **false** and b is **false** then $a \wedge b$ is **false**.
- If a is **false** and b is **chaos** then $a \wedge b$ is **false**.
- If a is **chaos** and b is **true** then $a \wedge b$ is **chaos**.
- If a is **chaos** and b is **false** then $a \wedge b$ is **chaos**.
- If a is **chaos** and b is **chaos** then $a \wedge b$ is **chaos**.

This is summarised in Table **2**.2.

$\wedge$	**true**	**false**	**chaos**
true	**true**	**false**	**chaos**
false	**false**	**false**	**false**
chaos	**chaos**	**chaos**	**chaos**

Table **2**.2: Conjunction: *"and"*, $\wedge$

2.5.3 Disjunction: "or", $\vee$

We proceed to explain the disjunction operator as it appears in $a \vee b$.

- If a is **true** and b is **true** then $a \vee b$ is **true**.
- If a is **true** and b is **false** then $a \vee b$ is **true**.
- If a is **true** and b is **chaos** then $a \vee b$ is **true**.
- If a is **false** and b is **true** then $a \vee b$ is **true**.
- If a is **false** and b is **false** then $a \vee b$ is **false**.
- If a is **false** and b is **chaos** then $a \vee b$ is **chaos**.

- If a is **chaos** and b is **true** then $a \vee b$ is **chaos**.
- If a is **chaos** and b is **false** then $a \vee b$ is **chaos**.
- If a is **chaos** and b is **chaos** then $a \vee b$ is **chaos**.

This is summarised in Table **2**.3.

$\vee$	**true**	**false**	**chaos**
true	**true**	**true**	**true**
false	**true**	**false**	**chaos**
chaos	**chaos**	**chaos**	**chaos**

Table **2**.3: Disjunction: *"or"*, $\vee$

2.5.4 Implication: "imply", $\supset$

- If a is **true** and b is **true** then $a \supset b$ is **true**.
- If a is **true** and b is **false** then $a \supset b$ is **false**.
- If a is **true** and b is **chaos** then $a \supset b$ is **chaos**.
- If a is **false** and b is **true** then $a \supset b$ is **true**.
- If a is **false** and b is **false** then $a \supset b$ is **true**.
- If a is **false** and b is **chaos** then $a \supset b$ is **true**.
- If a is **chaos** and b is **true** then $a \supset b$ is **chaos**.
- If a is **chaos** and b is **false** then $a \supset b$ is **chaos**.
- If a is **chaos** and b is **chaos** then $a \supset b$ is **chaos**.

This is summarised in Table **2**.4.

$\supset$	**true**	**false**	**chaos**
true	**true**	**false**	**chaos**
false	**true**	**true**	**true**
chaos	**chaos**	**chaos**	**chaos**

Table **2**.4: Implication: *"imply"*, $\supset$

You might wonder that why is $p \supset q$ **true** when p is **false.**

This is because the implication guarantees that when p and q are **true** then the implication is **true**. But the implication does not guarantee anything when the premise p is **false**. There is no way of knowing whether or not the implication is false since p did not happen. This situation is similar to the *"Innocent until proven Guilty"* stance, which means that the implication $p \supset q$ is considered **true** until proven **false**. Since we cannot call the implication $p \supset q$ q **false** when p is **false**, our only alternative is to call it **true**.

2.5.5 Equivalence: "if-and-only-if", $\equiv$

Without further ado:

$\equiv$	**true**	**false**	**chaos**
true	**true**	**false**	**chaos**
false	**true**	**true**	**chaos**
chaos	**chaos**	**chaos**	**chaos**

Table **2**.5: Equivalence: *"if-and-only-if"*, $\equiv$

2.5.6 Equality: "is equal to, is not equal to", $=, \neq$

The symbols $=$ and $\neq$ stand for relations.

Definition 56 Relation: *By a* relation *we shall, in this book, mean a mathematical concept which expresses the way in which two or more things are connected; a thing's effect on or relevance to another* ■

For the time being, when we have only formally introduced Boolean values and informally relied on numbers relations, like $=$ and $\neq$, connect either a pair of Boolean[value]s or a pair of numbers. Thus, in $a = b$, if a is the same [value] as is b, then $a = b$ is said to hold, to be **true**, otherwise **false**. If a stands for a Boolean and b for a number then $a = b$ makes no sense, and we say that $a = b$ stands for **chaos**.

2.6 The if .. then ... else ...end Conditional

Sections **2.5.1**–**2.5.4** informally introduced, in the *observer's language*, the **if .. then ... else ...end** construct. We "lift" that construct to be a construct of the *object language:*

- **if** $\mathcal{P}$ **then** $\mathcal{Q}$ **else** $\mathcal{R}$ **end**

where $\mathcal{P}$ is any Boolean expression and $\mathcal{Q}$ and $\mathcal{R}$ are, in general, any expression of the object language. In this chapter they are [just] Boolean expressions.

2.7 Proof Systems

2.7.1 Model Theory

Model theory was firs covered in Sect. **2.5** on page 54.

more to come

2.7.2 Axioms

Definition 57 Axiom: *An axiom, postulate, or assumption is a statement that is taken to be true, to serve as a premise or starting point for further reasoning and arguments. The word comes from the Ancient Greek word* $\alpha\xi\iota\omega\mu\alpha$ *(axōma), meaning that which is thought worthy or fit or that which commends itself as evident* ∎

more to come

2.7.3 Proof Theory

to come

2.7.4 Deduction Rules

https://personal.math.ubc.ca/~cytryn/teaching/scienceOneF10W11/handouts/OS.-proof.3inference.html

Definition 58 Deduction Rule: *In mathematical logic, a deduction theorem is a meta-theorem that justifies doing conditional proofs from a hypothesis in systems that do not explicitly axiomatize that hypothesis, i.e. to prove an implication* $A \supset B$*, it is sufficient to assume* A *as an hypothesis and then proceed to derive* B ∎

2.7.4.1 Modus Ponens

$$\frac{P \Rightarrow Q\ ; P}{\mathrm{Q}}$$

2.7.4.2 Modus Tollens

$$\frac{P \Rightarrow Q\ ;\ \sim Q}{\sim\ P}$$

2.7.4.3 Hypothetical Syllogism

$$\frac{P \Rightarrow Q\ ;\ Q \Rightarrow R}{P \Rightarrow R}$$

2.7.4.4 Disjunctive Syllogism

$$\frac{P \vee Q \ ; \ \sim P}{Q}$$

2.7.4.5 Constructive Dillemma

$$\frac{(P \Rightarrow Q) \ \wedge \ (R \Rightarrow S) \ ; \ P \ \vee \ R}{Q \ \vee \ S}$$

2.7.4.6 Destructive Dillemma

$$\frac{(P \Rightarrow Q) \ \wedge \ (R \Rightarrow S) \ ; \ \sim Q \vee \ \sim S}{\sim P \vee \ \sim T}$$

2.7.4.7 Simplification

$$\frac{P \wedge Q}{P}$$

2.7.4.8 Conjunction

$$\frac{P \ ; \ Q}{P \ \wedge \ Q}$$

2.7.4.9 Addition

$$\frac{P}{P \ \vee \ Q}$$

2.7.5 Proofs

$\boxed{\text{to come}}$

2.7.6 Proof Systems Summary

$\boxed{\text{to come}}$

2.8 Syntax

By *syntax* we shall mean the arrangement of elements (e.g., words or parts) and their composition (e.g., phrases or composite parts) to create well-formed structure (e.g., sentences or parts) in a language or model. [By words and phrases we mean those of a (written/spoken) laguages; and by parts we mean those of a domain model.]

We present, below, a first version of a syntax, a so-called `BNF Grammar`[8][9], of Boolean expressions, that is, expressions whose value are either of the Boolean truth values.

BNF Grammar: Boolean Expressions

```
<Boolean-expr>     ::=  true | false
                    |   (∼<Boolean-expr>)
                    |   (<Boolean-expr><InfixBoolean-op><Boolean-expr>)
                    |   b
                    |   ...
                    |   if <Boolean-expr>
                           then <Boolean-expr>
                           else <Boolean-expr> end
<InfixBoolean-op>  ::=  ∧ | ∨ |⊃|=|≠|≡
```

The `BNF Grammar` shall be understood as follows:

- **true** and **false** are a Boolean expressions.
- The pair $\sim$ followed by a Boolean expression of is a Boolean expression.
- The parenthesized triplet of two Boolean expressions separated by a Boolean operator, either $\wedge$, or $\vee$, or $\supset$, or $=$, or $\neq$, or $\equiv$, in that order, left-to-right, is a Boolean expression. then b is a Boolean expression.
- Let b be an identifier standing for a Boolean truth value, then b is a Boolean expression.
- There are many more kinds of Boolean expressions: over sets (Sect. **3.4.1** on page 68), numbers (Sect. **4.3.1** on page 79), Cartesians (Sect. **8.4** on page 113), lists (Sect. **10.3** on page 137), maps (Sect. **11.3** on page 147), etc.
- The **if-then-else-end** clause is a Boolean expression if the **then** and **else** clauses are Boolean expressions.

2.9 Summary and Conclusion

to be written

[8] https://en.wikipedia.org/wiki/Backus-Naur_form

[9] A `BNF Grammar`, syntactically, is a set of one or more `BNF Rules`. A rule has three elements: a left hand side so-called *non-terminal [syntax category] name*, shown within <...> brackets; a definition symbols: `::=`, and a right hand side syntax expression. The right hand side syntax expression consists of one or more syntax sub-expressions separated by vertical bars (|). A syntax sub-expression is a definite sequence of zero, one or more either a *terminal*, i.e., a literal (like **true, false, if, then** or **else**), or *non-terminal names* – identifiers not "surrounded" by <...>. Semantically a `BNF Grammar` defines, from one point of view, a possibly infinite set of texts consisting of terminals.

2.9.1 Summary

It is all very simple: We have introduced

- *truth values* **true** and **false**, Sect. **2.2.1**;
- the *Boolean* **type Bool** ,Sect. **2.2.2**;
- the *Boolean operations* denoted by the *Boolean operators* $\sim, \vee, \wedge, \supset, =, \neq$ and $\equiv$, Sect. **2.5**;
- the *model theory truth tables*, Sect. **2.4**; and
- the **if ... then ... else ... end** MoLa language clause, Sect. **2.6**.

The less simple things were: in Sect. **2.7**, *Proof Systems*:

- More on *Model Theory*, Sect. **2.7.1** on page 58;
- on *Axioms*, Sect. **2.7.2** on page 58;
- on *Proof Theory*, Sect. **2.7.3** on page 58;
- on *Deduction Rules*, Sect. **2.7.4** on page 58; and
- on *Proofs*, Sect. **2.7.5** on page 59.

Our coverage of the former, "the simple", is sufficient for this book. Our coverage of the later, "the less simple", is only presented here in order to entice the curious and capable reader onto real *mathematical Logic!*

2.9.2 Conclusion

Mathematics may, by some, be called *"The Queen of Sciences"*[10]. We may agree, but add: *"Mathematical Logic is the first, primus inter pares*[11]*, among the mathematical disciplines"* !

We refer the student to some seminal textbooks in mathematical logic.

- **Under-graduate Texts:**
  - Smullyan, Raymond: A Beginner's Guide to Mathematical Logic, Dover Books on Mathematics.ISBN 0486492370;
  - Enderton, Herbert (2001). A mathematical introduction to logic (2nd ed.). Boston MA: Academic Press. ISBN 978-0-12-238452-3;
  - Mendelson, Elliott (1997). Introduction to Mathematical Logic (4th ed.). London: Chapman & Hall. ISBN 978-0-412-80830-2;

[10]– where *"Physics is the King of Sciences* – with the *Queen*, i.e., *Mathematics*, supporting all the sciences.
[11]– first among equals

  - van Dalen, Dirk (2013). Logic and Structure. Universitext. Berlin: Springer. doi:10.1007/978-1-4471-4558-5. ISBN 978-1-4471-4557-8;

  - van Dalen, Dirk (2013). Logic and Structure. Universitext. Berlin: Springer. doi:10.1007/978-1-4471-4558-5. ISBN 978-1-4471-4557-8.

- **Under-graduate Texts:**

  - Barwise, Jon, ed. (1989). Handbook of Mathematical Logic. Studies in Logic and the Foundations of Mathematics. Amsterdam: Elsevier. ISBN 9780444863881;
  - Kleene, Stephen Cole.(1952), Introduction to Metamathematics. New York: Van Nostrand. (Ishi Press: 2009 reprint);
  - Kleene, Stephen Cole. (1967), Mathematical Logic. John Wiley. Dover reprint, 2002. ISBN 0-486-42533-9;
  - Shoenfield, Joseph R. (2001) [1967]. Mathematical Logic (2nd ed.). A .K. Peters; ISBN 9781568811352.

2.10 Exercises

Exercise 9 **XLogic:**

Exercise 10 **YLogic:**

Exercise 11 **ZLogic:**

Chapter 3

Sets

Contents

a Band of Musicians	a Bevy of Beauties
a Bunch of Crooks	a Crew of Sailors
a Deck of Cards	a Fleet of Ships
a Flock of Geese	a Gang of Outlaws
a Group of People	a Herd of Cattle
a Mop of Hair	a Pack of Dogs
a Posse of Vigilantes	a Pride of Lions
a School of Dolphins	a Suite of Bells
a Swarm of Flies	a Volley of Arrows

– *all are examples of sets of entities of a specified type*

Motivation: Sets

Around us, in our world, in any universe, sets abound – as the above text illustrate. The concept of sets can be said to be a universal: can be justified by rational arguments, cf. [83–87]. There is no way around it, we cannot escape the notion of sets when creating domain models. [We are not using sets because they happen to be a foundation for mathematics. It's the other way around: Mathematics cannot escape dealing with sets!]

Study Hint: *This chapter should be studied carefully. Sections* **3.7**–**3.8** *are more for reference – for the mature, capable reader.*

In this chapter we shall introduce the programming cum modelling language MoLa's concept of **Sets**. The German mathematicians Ernst Zermelo (1871–1953) and Abraham Fraenkel (1891–1965) made major contributions to the mathematical concept of sets [1, 44, 45].

The present chapter and chapters **8**, **10** and **11**, are similarly structured[1].

3.1 Informal Presentations of Sets: Venn Diagrams

Figure **3**.1 on the facing page shows five *Venn*[2] *Diagrams*. You should think of the shaded areas to "contain" a number of [set] elements. Consider, for example, the leftmost diagram. You are to interpret, to understand the two shaded circles, labeled **A** and **B**, as designating, standing for possibly infinite sets of entities. Some entities of set **A** may also be entities of set **B**.

The *set union*, *cup*, of **A** and **B**, i.e., **A**∪**B** [see the second diagram from the left in Fig. **3**.1], is then the set of elements of either **A** or **B** or both.

The *set intersection*, *cup*, of **A** and **B**, i.e., **A**∩**B** [see the third diagram from the left in Fig. **3**.1], is then the set of elements of both **A** and **B**.

[1] **Editorial note:** But not yet in the "similarly structured" form that I would wish!

[2] John Venn, FRS, FSA (1834–1923) was an English mathematician, logician and philosopher noted for introducing Venn diagrams, which are used in logic, set theory, probability, statistics, and computer science [Wikipedia].

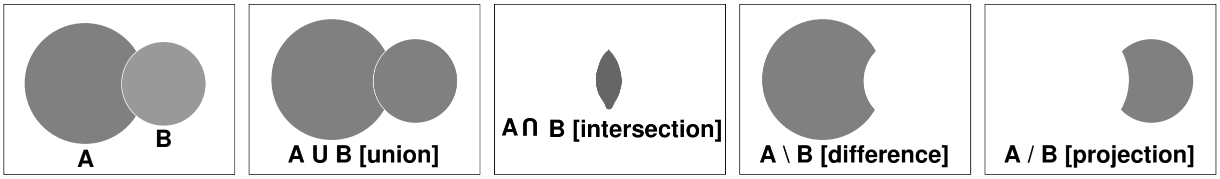


Figure 3.1: Two sets, set union, intersection, difference \, and complement /

The *set difference*, \, of **A** and **B**, i.e., **A****B** [see the fourth diagram from the left in Fig. **3**.1], is then the set of elements of **A** which are not elements of **B**.

The *set complement*, /, of **A** and **B**, i.e., **A**/**B** [see the fifth diagram from the left in Fig. **3**.1], is then the set of elements of **B** which are not elements of **A**.

Definition 59 Set: *By a set we shall, loosely, understand an unordered collection of distinct elements (i.e., entities) — something for which it is meaningful to speak about the following operations: (i) an entity being a member of a set (or not),* $\in$*; (ii) the union (merging) of two or more sets into a set (of all the elements of the argument sets),* $\cup$*; (iii) the intersection of two or more sets into a set (of those elements which are in all argument sets)* $\cap$*; (iv) the complement of one set with respect to another set* \ *or* /*; (v) whether one set is a proper subset or just a subset of another set,* $\subset$ *and* $\subseteq$*, or whether they are equal or not,* $=$*, resp.* $\neq$*; and (vi) the cardinality of a (finite) set (i.e., how many members it "contains")* **card***; etc.* ■

3.2 Set Presentations

The present section is structured into two subsections, as are Sects. **10.1** and **11.1**.

3.2.1 Set Enumeration

Let a_1, a_2, ..., a_n, ... stand for some distinct elements, then

value { a_1, a2, ..., a_n }, respectively { a_1, a2, ..., a_n, ... }

expresses an informal, abstract way of *explicitly enumerating* a finite, respectively infinite sets of n element. It is the use of further unidentified a_is and the ellipses: "..." that makes the presentation informal and abstract. The first ellipses to "abstract" from having to enumerate all a elements, the second ellipses to indicate infinity. A formal, concrete, finite set example could be:

Example 37 A Basket of Fruits: Let a_1, a_2, a_3 stand for three distinct $a$pples, p_1, p_2 for two distinct $p$ears, and o for a single orange, then

value { a_1, p_2, a_3, a_2, o }

then exemplifies a basket, a finite set, of fruits ■

3.2.2 Set Comprehension

Definition 60 **Set Comprehension:** *By a set comprehension we shall mean the description of a possibly infinite set in terms of a predicate that the comprehended set elements must satisfy* ∎

Exercise 12 **A Set Example:** *Let fact name the factorial function, then*

{ fact(i) | i:**Nat** • i ∈ {1..6} }

expresses a simple set of six elements, the first six factorials. Here the predicate is i:**Nat** • i ∈ {1..6} ∎

Let A be some type with elements a_1, a_2, ..., a_n, ... and let aset be a finite or infinite set of element in A. Let p:P be a predicate over elements of A, and let q:Q be a function over [perhaps not all] a:A yielding elements b of type B. Then the last line in the below five lines of two MoLa *specification units*[3]

type
    A, B
value
    aset:A-**set**, p: A→**Bool**, q: A $\stackrel{\sim}{\to}$ B
    qaset:B-**set** = { q(a) | a:A, aset:A-**set** • a ∈ aset ∧ p(a) }

expresses a *set comprehension*. In short: it denotes the set of all those q(a) of type B for which, "such that" (|), the a, of type A, is in aset and the property p(a) holds. The comprehended sets may be finite or infinite!

The signs { and } can be said to form and delineate the set. The | separates the text between the { and } into two texts. To the left of | is an expression, here just a. To the right of | there are two texts separated by a •. Between | and • the clause defines the type of a, hence its "larger" range, and its actual range aset. Between • and }, the as are limited here to within aset, and the predicate clause, p(a), delimits the as to those which satisfy that predicate, i.e., for which as holds, i.e,, is **true**.

Set comprehension, in general, usually applies to a set s, of elements of type, say A. Comprehension then results in a set, say, of type B elements.

These latter elements, q(a), derive from such s elements, a, which satisfy some predicate, $\mathcal{P}$(a). The resulting elements, p(a) follows .

3.3 Set Types

Let [the arbitrarily chosen, capital letter identifier] A name a type of *entities*, and let [the likewise arbitrarily chosen, capital letter identifier] B name the type of sets of A entities, then the three MoLa *specification units*

[3] cf. definition 16 on page 9.

- [1:] **type** A
- [2:] **type** $B_f = A$-**set**
- [3:] **type** $B_i = A$-**infset**

introduces type A and *defines* B_f and B_i. B_f, respectively B_i define collections of *finite cardinality* respectively possibly *infinite cardinality* sets of A entities. By *cardinality* we mean the quantity of entities of a set.

The above type *introduction* and type *definitions* shall be understood as follows:

- [1:] There is the literal, the keyword, **type**.
  - A is [a perhaps arbitrarily chosen] *type identifier*;
  - here it denotes some further undefined space of values.
- [2–3:] There is the literal, the keyword, **type** and an equal symbol, =.

  Together they "signal", to the reader, that what follows is the definition of a type.

  - with left-hand side type identifiers, here B_f and B_i, being the names of the type being defined,
  - with a right-hand side *type expression*, here A-**set**, respectively A-**set**.
  - [2.] B_f defines a space of finite sets of elements of A – even though A may contain an infinite number of elements.
  - [3.] B_i defines a space of both finite and infinite sets of elements of A – where, if A is a space of finite "size", then B_i will not contain infinite sets!

3.4 Model Theory: Sets

Let $e_1, e_2, \ldots, e_n$ be arbitrary elements (i.e., mathematical entities). Let us assume, without loss of generality (of what we shall have to say next), that they are all distinct and elementary, i.e., atomic. That is, no e_i involve functions, or other sets, etc. Then when writing $\{e_1, e_2, \ldots, e_n\}$ we mean the set, which we may name s, of n distinct elements e_i for $i = 1 \ldots n$. $\{\}$ designates the empty set (of no elements). $\{$ and $\}$ are the set-forming braces.

We take membership, $\in$, of a set, $e \in s$, to be a further unexplained primitive function. $e \in s$ holds, i.e., is **true,** if e is one of the e_i for $i = 1 \ldots n$. Otherwise the expression $e \in s$ is **false.**

Based on the membership function we can now define[4] the standard collection of operations over sets. Let e, s, s' designate any element and any two sets.

[4]The below explications, **3**.1–**3**.8, are all in the classical style of mathematics.

3.4.1 Set Operations

First we introduce and explain set-forming operations. The operation names, i.e., the set forming operators are: $\cup, \cap, \setminus$ and $/$.

3.1. The **union**, $\cup$, of two sets, s and s', is the set of entities e which are *either* in s or ($\vee$) in s' [or in *both*].

3.2. The **intersection**, $\cap$, of two sets, s and s', is the set of entities e which are *both* in s and ($\wedge$) in s'.

3.3. The **difference**, $\setminus$, of two sets, s and s', is the set of entities e which are in s but not ($\notin$) in s'.

3.4. A set is a **projection**, $/$, of a set s' *with respect to* another set s, if its entities e are in s' *and not in* ($\notin$) s.

In mathematics, not MoLa:

$$s \cup s' = \{e \mid e \in s \vee e \in s'\} \quad (\mathbf{3}.1)$$
$$s \cap s' = \{e \mid e \in s \wedge e \in s'\} \quad (\mathbf{3}.2)$$
$$s \setminus s' = \{e \mid e \in s \wedge e \notin s'\} \quad (\mathbf{3}.3)$$
$$s \ / \ s' = \{e \mid e \in s' \wedge e \notin s\} \quad (\mathbf{3}.4)$$

3.4.2 Set Relations

First we introduce and explain set relations. The relation names, i.e., the set forming operators are: $\subset, \subseteq, =$ and $\neq$.

3.5. A set s is a **proper subset** ($\subset$) of another set s' if *for all* ($\forall$) entities e in s *implies* ($\Rightarrow$) that e is also in s', *and* ($\wedge$) there *exists* ($\exists$) an entity e' in s' which is *not in* ($\notin$) s.

3.6. A set s is a **subset** ($\subseteq$) of another set s' if *for all* ($\forall$) entities e in s *implies* ($\Rightarrow$) that e is also in s'.

3.7. Two sets s and s' are **equal** ($=$) if they are *subsets* of each other.

3.8. Two sets s and s' are **not equal** ($\neq$) if they are *not equal*.

In mathematics, not MoLa:

$$s \subset s' = \forall e \bullet e \in s \Rightarrow e \in s' \wedge \exists e \bullet e \in s' \wedge e \notin s \quad (\mathbf{3}.5)$$
$$s \subseteq s' = \forall e \bullet e \in s \Rightarrow e \in s' \quad (\mathbf{3}.6)$$
$$s = s' = s \subseteq s' \wedge s' \subseteq s \quad (\mathbf{3}.7)$$
$$s \neq s' = \neg(s = s') \quad (\mathbf{3}.8)$$

3.4.3 Set Arithmetics

Presupposing knowledge of natural numbers, see Chapter **4**, we can define the **card**inality of a finite set as the number of its [distinct] entities: the **card**inality of the empty set is 0; and the **card**inality of a non-empty set s consisting of an entity e and a set s' [of which e is not an element) is 1 plus the **card**inality of s'.

$$\begin{aligned} \textbf{card}(\{\}) &= 0 && (\mathbf{3}.9) \\ \textbf{card}(\{e\} \cup s) &= 1 + \textbf{card}(s) && (\mathbf{3}.10) \end{aligned}$$

3.4.4 Relations

Let X stand for any set and R for a relations relations over sets. From `https://en.wikipedia.org/wiki/Relation_(mathematics) [Wikipedia]` we *"unabashedly lift"*:

- **Reflexive:** for all $x \in X, xRx$. For example, $\geq$ is a reflexive relation but $>$ is not.
- **Irreflexive:** for all $x \in X$, $\neg xRx$. For example, $>$ is an irreflexive relation, but $\geq$ is not.
- **Symmetric:** for all x, y, zX, if xRy then yRx. For example, "is a blood relative of" is a symmetric relation, because x is a blood relative of y if and only if y is a blood relative of x.
- **Antisymmetric:** for all $x, y \in X$, if xRy and yRx then $x = y$. For example, $\leq$ is an antisymmetric relation; so is $>$, but vacuously so (the condition in the definition is always false).
- **Asymmetric:** for all $x, y \in X$, if xRy then $\neg yRx$. A relation is asymmetric if and only if it is both antisymmetric and irreflexive. For example, $>$ is an asymmetric relation, but $\geq$ is not.
- **Transitive:** for all $x, y, z \in X$, if xRy and yRz then xRz. A transitive relation is irreflexive if and only if it is asymmetric. For example, "is ancestor of" is a transitive relation, while "is parent of" is not.
- **Equivalence:** A relation that is reflexive, symmetric, and transitive. It is also a relation that is symmetric, transitive, and serial, since these properties imply reflexivity.
- **Partial Order:** A relation that is reflexive, antisymmetric, and transitive.
- **Strict Partial Order:** A relation that is irreflexive, antisymmetric, and transitive.
- **Total Order:** A relation that is reflexive, antisymmetric, transitive and connected.
- **Strict Total Order:** A relation that is irreflexive, antisymmetric, transitive and connected.

3.5 Playing Around with Sets

In this and in Sects. **4.4**, **8.5**, **10.4**, and **11.4**, we present a number of "standard" examples of operations on sets, numbers, Cartesians, lists and maps. In this section with sets.

3.5.1 Classical Functions of Sets

Example 38 **Power Set:**

25. Let A be any non-function type, and
26. let sa:SA be any [finite] set of A elements.
27. The power set of sa, psa, is a set of sets of A elements such that any and only subsets of sa, including the empty subset are in psa.

type
25. A
26. SA = A-**set**
27. PSA = SA-**set**
value
26. sa:SA
27. power_set: SA → PSA
27. power_set(sa) ≡ { ssa | ssa:SA • ssa ⊆ sa }

Example 39 **Set Union Closure:** A collection, or family, of sets is considered *union-closed* if the union of any two sets in the family equals any existing set in the family[5].

28. Let A be any type and sa be any [finite] set of elements of A.
29. The function set_union_closure, applied to any sa yields the set union closure of sa.

type
28. A
value
28. sa:A-**set**
29. set_union_closure: A-**set** → A-**set**
29. set_union_closure(sa) ≡ { ssa1 ∪ ssa2 | ssa1,ssa2:A-**set** • ssa1 **in** sa ∧ ssa3 **in** sa }

Example 40 **The Partitioning Problem:** A classical problem in management is the *partitioning problem*.

[5]https://www.quantamagazine.org/long-out-of-math-an-ai-programmer-cracks-a-pure-math-problem-20230103/?mc_cid=ca0d0b5110&mc_eid=783b63461a

30. Let A be any non-function type,
31. S be the type of finite sets of $a : A$, and
32. P be the type of finite sets of elements of S.
33. $p : P$ *is a partition of* $s : S$,
    (a) if the union of all sets in p is s,
    (b) if for all s', s'' in S both are subsets of s and both are in p,
    (c) and they have no elements in common.
34. *Partitions* of $s : S$ are maximal sets
    (a) of partitions such there there
    (b) does not exist a $p' : P$
    (c) not in ps and
    (d) $p' : P$ *a partition of* s.

```
type
30.   A
31.   S = A-set
32.   P = S-set
value
33.   is_partition: P × S → Bool
33.   is_partition(p,s) ≡
33a.     ∪ p = s ∧
33b.     ∀ s′,s″:S • s′⊆s ∧ s″⊆s ∧ s′ ∈ p ∧ s″ ∈ p
33c.        ⇒ s′ ∩ s″ = {}

34.    partitions: S → P-set
34.    partitions(s) as ps
34a.       • ∀ p:P • p ∈ ps ⇒ is_partition(p,s)
34b.          ∧ ∼∃ p′:P •
34c.                p′ ∉ ps ∧
34d.                is_partition(p′,s)  ∎
```

3.5.2 Functions over Sets

3.5.2.1 Sum of Number Sets

35. Let S be the type of finite sets of natural numbers.
36. Then function sum_of_numbers sum the numbers of sets s S:
    (a) The **cases** construct examines set s.
    (b) If it is empty, $\{\}$, then the sum is 0;
    (c) If it is not empty, then s can be expressed as the union, $\cup$, of a singleton set of a number n and the set s′ such that their union is s, in shich case the sum of s is the sum of n and the sum of s′.

type
35. S = **Nat-set**
value
36. sum_of_numbers: S $\to$ **Nat**
36. sum_of_number(s) $\equiv$
36a. **case** s **of**:
36b. $\{\} \to 0$,
36c. $\{n\} \cup s' \to n +$ sum_of_number(s')
36. **end**

Another other version of sum_of_numbers is:

value
36.′ sum_of_numbers(s) $\equiv$
36a.′ **if** $s=\{\}$
36b.′ **then** 0
36c.′ **else let** n:**Nat** • $n \in s$ **in** n + sum_of_numbers($s \backslash \{n\}$)
36.′ **end**

3.5.2.2 Equal Sum Partitions of Number Sets

37. Let PS be set of sets of natural numbers.
38. An *equal sum partition set of numbers* is a partition, on ps:PS, all of whose element sets have the same sum.

type
37. PS = (**Nat-set**)-**set**
value
38. equal_sum_part_set: PS $\to$ **Bool**
38. equal_sum_part_set(ps) $\equiv$
38. is_partition(ps) $\wedge \forall$ s,s′:**Nat-set**•$\{s,s'\} \subseteq$ps $\wedge$ sum_of_numbers(s)=sum_of_numbers(s′)

3.5.2.3 A Sociology Example

39. Let c:C stand for a citizen c of type C.
40. Let g:G stand for a group g of citizens of type of G.
41. Let s:S stand for any set of groups, respectively the type of all such.
42. Two otherwise distinct groups are related to one another if they share at least one citizen, the liaisons, l:L.

43. A network n:N is a set of groups such that for every group in the network one can always find another group with which it shares liaisons.

Solely using the set data type and the concept of subtypes, we can model the above:

type
39. C
40. $\mathsf{G}' = \mathsf{C}\text{-}\mathbf{set},\ \ \mathsf{G} = \{|\ \mathsf{g}{:}\mathsf{G}' \bullet \mathsf{g}{\neq}\{\}\ |\}$
41. $\mathsf{S} = \mathsf{G}\text{-}\mathbf{set}$
42. $\mathsf{L}' = \mathsf{C}\text{-}\mathbf{set},\ \ \mathsf{L} = \{|\ \ell{:}\mathsf{L}' \bullet \ell{\neq}\{\}\ |\}$
43. $\mathsf{N}' = \mathsf{S},\ \ \mathsf{N} = \{|\ \mathsf{s}{:}\mathsf{S} \bullet \mathsf{wf\_S(s)}\ |\}$
value
41. $\mathsf{wf\_S}\text{: } \mathsf{S} \rightarrow \mathbf{Bool}$
41. $\mathsf{wf\_S(s)} \equiv \forall\ \mathsf{g}{:}\mathsf{G} \bullet \mathsf{g} \in \mathsf{s} \Rightarrow \exists\ \mathsf{g}'{:}\mathsf{G} \bullet\ \mathsf{g}' \in \mathsf{s} \wedge \mathsf{share(g,g')}$
42. $\mathsf{share}\text{: } \mathsf{G}{\times}\mathsf{G} \rightarrow \mathbf{Bool}$
42. $\mathsf{share(g,g')} \equiv \mathsf{g}{\neq}\mathsf{g}' \wedge \mathsf{g} \cap \mathsf{g}' \neq \{\}$
42. $\mathsf{liaisons}\text{: } \mathsf{G}{\times}\mathsf{G} \rightarrow \mathsf{L}$
42. $\mathsf{liaisons(g,g')} = \mathsf{g} \cap \mathsf{g}'\ \mathbf{pre}\ \mathsf{share(g,g')}$

Annotations: L stands for proper liaisons (of at least one liaison). G', L' and N' are the "raw" types which are constrained to G, L and N. $\{|\ \mathsf{binding{:}type\_expr} \bullet \mathsf{bool\_expr}\ |\}$ is the general form of the subtype expression. For G and L we state the constraints "in-line", i.e., as direct part of the subtype expression. For N we state the constraints by referring to a separately defined predicate. wf_S(s) expresses — through the auxiliary predicate — that s contains at least two groups and that any such two groups share at least one citizen. liaisons is a "truly" auxiliary function in that we have yet to "find an active need" for this function!

The example is a model of a social structure suggested by [89, **Fei XaioTong**'s book *From the Soil* – The Foundations of Chinese Society: *XiangTu ZhongGuo*][6].

3.6 Syntax

By *syntax* we shall mean the arrangement of elements (e.g., words or parts) and their composition (e.g., phrases or composite parts) to create well-formed structure (e.g., sentences or parts) in a language or model. [By words and phrases we mean those of a (written/spoken) laguages; and by parts we mean those of a domain model.]

We present, below, a second version[7] of a syntax, a so-called `BNF Grammar`[8], of set

[6] https://en.wikipedia.org/wiki/Fei_Xiaotong
[7] Section **2.8** on page 59 brought a first version
[8] https://en.wikipedia.org/wiki/Backus-Naur_form

expressions, that is, expressions whose value are sets[9].

BNF Grammar: Set Expressions

<Set-Expr>	::=	{}
	|	{ <Expr-sequence> }
	|	{ <Id> | <IdTypList> • <Pred-expr> }
	|	<Set-Expr> <Set-op> <Set-Expr>
<IdTypList>	::=	<Id>:<Type-expr>
	|	<IdTypList>, <Id>:<Type-expr>
<Set-op>	::=	∪ | ∩
<Expr-sequence>	::=	<Constant>
	|	<Variable>
	|	<Expression> , <Expr-sequence>
<Type-expr>	::=	...
<Pred-expr>	::=	...
<Constant>	::=	...
<Variable>	::=	<Id>
<Boolean-expr>	::=	<Set-Expr> <Set-relation> <Set-Expr> | ...
<Set-relation>	::=	=|≠|⊂|⊆
<Arith-expr>	::=	**card** <Set-Expr> | ...

The `BNF Grammar` above also shows rudiments of the syntax of Boolean and arithmetic expressions (...).[10]

3.7 The Zermelo Fraenkel Axiom System for Sets

We "lift" from `https://mathworld.`**wolfram**`.com/`**Zermelo-FraenkelAxioms**`.html`.

The Zermelo-Fraenkel axioms are the basis for *Zermelo-Fraenkel set theory*. In the following, $\exists$ stands for "there exists", $\forall$ means "for all", $\in$ stands for "is an element of", $\{\}$ for "the empty set", $\supset$ for "implies", $\wedge$ for "and", $\vee$ for "or", $\equiv$ for "if-and-only-if" (identity), and $=$ for "is equal to". By $\bullet$ we mean: "it is the case that".

- **Axiom of Extensionality:** If X and Y have the same elements, then $X = Y$.

  $\forall\ u \bullet\ (u \in X \equiv u \in Y) \supset\ X = Y.$

- **Axiom of the Unordered Pair**[11]**:** For any a and b there exists a set $\{a, b\}$ that contains exactly a and b.

  $\forall a, \forall b, \exists c, \forall x \bullet\ (x \in c \equiv (x = a \vee x = b))$

- **Axiom of Subsets**[12]**:** If ϕ is a property

[9] $\{,\},\cup,\cap,=,\neq,\subset,\subseteq,\bullet$, and | (when occurring in a right hand side syntax expression) are terminal names and literals.

[10] We shall show `BNF Grammars` for type expressions, predicate expressions and constants elsewhere.

[11] Also called **Axiom of Pairing**

[12] Also called **Axiom of Separation** or **Axiom of Comprehension**

(with parameter p), then for any X and p there exists a set $Y = \{u{\in}X \bullet \phi(u,p)$ that contains all those u in X that have the property ϕ.

$\forall X$, $\forall p$, $\exists Y$, $\forall u$
$\bullet$ $(u{\in}Y \equiv (u \in Z \wedge \ \phi(u,p)))$

- **Axiom of the Sum Set**[13]**:** For any X there exists a set $Y{=} \cup X$, the union of all elements of X.

  $\forall X, \exists Y, \forall u$
  $\bullet$ $(u \in Y \exists z \bullet \ (z \in X \wedge u \in z))$

- **Axiom of the Power Set:** For any X there exists a set $Y = \mathcal{B}(X)$, the set of all subsets of X.

  $\forall X$,$\exists$Y,$\forall u\bullet$ $(u{\in}Y{\equiv}u \subseteq X)$

- **Axiom of Infinity:** There exists an infinite set.

  $\exists S \bullet \ S = \{\} \wedge \ \forall x \in S \bullet \ (x \cup \ \{x\} \in S)$

- **Axiom of Replacement:** If F is a function, then for any X there $\exists$ a set $Y = F[X] = \{F(x) : x \in X\}$.

  $\forall x, y, z \bullet$
  $(\phi(x,y,p) \wedge \ (\phi(x,z,p) \supset (y = z))) \ \supset$
  $\forall X, \exists Y, \forall y \ \bullet (y \in Y)$

- **Axiom of Foundation**[14]**:** Every nonempty set has an $\in$-minimal (inclusion-minimal) element.

  $\forall S \bullet \ (S \neq \{\}) \supset (\exists x \in S \bullet S \cap x = \{\})$

- **Axiom of Choice:** Every family of nonempty sets has a choice function A.

  $\forall x \in a$,$\exists A(x,y) \supset \exists y$,$\forall x \in a \supset A(x, y(z))$

The above system of axioms without the axiom of choice is called Zermelo-Fraenkel set theory, denoted **ZF.** This system of axioms minus the axiom of replacement (and choice) is called Zermelo set theory, denoted **Z.** The set of all axioms (i.e., with the axiom of choice) is usually denoted **ZFC**.

3.8 Proofs

to be written

3.9 Closing

to be written

3.9.1 Summary

It is all either very simple.

We have covered a number of set concepts:

- *Set Presentations*: Enumeration and comprehension, Sect. **3.2**;

[13] Also called **Axiom of Union**
[14] also called **Axiom of Regularity**

- *Set Types*: **type** A-**set** and A-**infset**, Sect. **3.3**; and
- *Model Theory: Sets*: $\in, \cup, \cap, \backslash, /, =, \neq, \subset, \subseteq$ and **card**, Sect. **3.4**.

or less so:

- *The Zermelo Fraenkel Axiom System for Sets*, Sect. **3.7**; and
- *Proofs*, Sect. **3.8**.

3.9.2 Conclusion

We refer the interested, mature reader to seminal textbooks on set theory.

- P.R. Halmos: Naive Set Theory (1974);
- AA Fraenkel, Y Bar-Hillel, A Levy: Foundations of set theory (1973);
- Suppes: Axiomatic set theory (1972) ;
- Quine: Set theory and its logic (1969); and
- Herbert B. Enderton: Elements of Set Theory (1977).

3.10 Exercises

Exercise 13 XSets: *[We refer to Sect.* **??** *on page ??.]*

Exercise 14 YSets: *[We refer to Sect.* **??** *on page ??.]*

Exercise 15 ZSets: *[We refer to Sect.* **??** *on page ??.]*

Chapter 4

Numbers and Numerals

Contents

In this chapter we shall introduce the programming cum modelling language MoLa's concepts of **numbers** and **numerals**.

Motivation: Numbers

Numbers can be motivated by a philosophical argument: First there are sets. Again, sets can also be motivated by a philosophical argument. Sets were covered already in Chapter **3**. Given sets we can speak of the empty set, $\{\}$, of no members, corresponding to the natural number 0, and of the empty set of just one member, say $\{\{\}\}$, corresponding to the natural number 1. And so forth.

Study Hint: *Sections* **4.1**–**4.4** *provide basic material, Sect.* **4.5** *is more advanced.* $\boxed{\textit{more to come}}$

4.1 Number Presentations

4.1.1 Informal

We can speak of **Natural numbers:** the numbers 0, 1, 2, etc.; **Integers:** the numbers ..., -2, -1, 0, 1, 2, ..., etc.; and **Rational:** those that results from the division of one integer by a non-zero integer. There are other types of numbers, like *irrational*, *complex* and *transcendental* numbers; but we shall not need them here.

We informally represent the natural numbers by the **numerals:** 0, 1, 2, etc., as You known them best, in the radix[1] 10 decimal number system.

4.1.2 Radix n Numerals

There are many ways of representing numbers.

In *the decimal numeral system* 10 characters, usually the characters, or digits, 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9, are used as follows: Let x_i stand for digits in $x_n...x_1x_0$, where $x_n...x_1x_0$ stand for a numeral, then $x_n...x_1x_0$ denote the number $x_n * 10^n + ... + x_1 * 10^1 + x_0 * 10^0$.

In a *radix* r *numeral system* the characters, say, a, b, ..., c are used as follows: Let x_i stand for radix r characters in $x_n...x_1x_0$, where $x_n...x_1x_0$ stand for a numeral, then $x_n...x_1x_0$ denote the number $x_n * r^n + ... + x_1 * r^1 + x_0 * r^0$.

Let r be "2", i.e., the binary system. The two "binary digits" are usually expressed by $\bigcirc$ and $|$. Interpret $\bigcirc$ as 0 and $|$ as 1. $|\bigcirc|||\bigcirc$ is then the number also designated by $32 + 8 + 4 + 2 = 46$.

[1]**radix:** In a positional numeral system, the radix or base is the number of unique digits, including the digit zero, used to represent numbers. For example, for the decimal system (the most common system in use today) the radix is ten, because it uses the ten digits from 0 through 9.

4.2 Types

We refer to the type of *natural numbers* by the type name (a literal[2]) **Nat**, *integers* by the type name (a literal) **Int**, and *real* by the type name (a literal) **Real**. They relate as follows:

- **Nat** $\subset$ **Int** $\subset$ **Real**,

where $\subset$ here stands for type inclusion: all natural numbers form a proper subset of all integers, and all integers form a proper subset of all reals. The $\subset$ operator/operation, as used here, is not in a MoLa expression. The **Nat** $\subset$ **Int** $\subset$ **Real** clause is a clause of the meta language, here mathematics, wit which we explain MoLa.

4.3 Operations on Numbers

4.3.1 The Operations

Let a, b and r stand for numbers; r for reals, e.g., by being numerals, c for positive reals, let n be a natural number. We take for granted the following operations on numbers:

- $a + b$: $+$ stands for the binary operation of addition of numbers;
- $-a, a - b$: $-$ either stands a unary operation that "negates" a number, or it stands for the binary operation that subtracts the right operand number from the left operand number;
- $a * b$: $*$ stands for the binary operation of multiplication of numbers;
- a/b: $/$ stands for the binary operation of division of numbers;
- a^n: stands for the exponential of a wrt. n;
- $a < b$: a is properly less than b;
- $a \leq b$: a is less than or equal to b, b different from 0;
- $a = b$: a is equal to b;
- $a \neq b$: a is not equal to b;
- $a \geq b$: a is greater than or equal to b;
- $a > b$: a is properly greater than b;
- $\lfloor r \rfloor$: [*floor*] the greatest integer less than or equal to r;

[2] **literal:** true to fact; not exaggerated; actual or factual

- $\lceil r \rceil$: [*ceil*] the least integer greater than or equal to r;
- $\sqrt{(c)}$: the square root of the positive rational number c;
- $a{\uparrow}b$: the exponentiation of a to the power b, i.e., $a{\uparrow}b = a * a \cdots a$, b times;
- $log_{10}(r)$: Logarithm is inverse function to exponentiation. The logarithm of a number r to the base 10 is the exponent to which 10 must be raised to produce r.

4.3.2 Operation Types

Let add, min, sub, mpy, div, exp, and max name the addition, negation, subtraction, multiplication, division, exponentiation and absolute value operations. We can **type**[3] the values of these operations. Let NUM stand for the type of numbers, whether natural, integers or reals.

value
add,sub,mpy,exp: NUM×NUM→NUM
div: NUM×NUM$\stackrel{\sim}{\to}$NUM
max: NUM→NUM

We shall have more to say about numbers and their operations in the next chapter.

4.3.3 Unary and Binary Operations

- The arithmetic operations of $\sqrt{.}$, $\mathbf{log}_{10}(.)$, $\lceil . \rceil$, $\lfloor . \rfloor$ and $-$ are unary: apply to one argument; $\lceil . \rceil$ and $\lfloor . \rfloor$ are *distributive-fix*; $-$ is a prefix operator.
- The arithmetic operations of $+, -, *, /$ and $\uparrow$ are binary: apply to two arguments, and are all *infix*.

4.3.4 Commutative, Associative and Distributive Properties

We introduce the concepts of *commutative*, *associative*, and *distributive* laws.

- **Commutative Law** implies: $a\mathbf{o}b = b\mathbf{o}a$ for all numbers a, b. Operations $+$ and $*$ are commutative.
- **Associative Law** implies: $a\mathbf{o}(b\mathbf{o}c) = a\mathbf{o}(b\mathbf{o}c)$ for all numbers a, b, c. Operations $+$ and $*$ are associative.
- **Distributive Law** implies: $a\mathbf{o}_{mult}(b\mathbf{o}_{plus}\mathrm{c}) = (a\mathbf{o}_{plus}b)\mathbf{o}_{sum}(a\mathbf{o}_{mult}c$ for all numbers a, b, c. Operation $*$ [$\mathbf{o}_1$] *distributes* of addition $(+)$ [$\mathbf{o}_2$].

[3]Not type, as on a keyboard, by name the class of values that, in this case, the arithmetic functions stand for

4.4 Playing Around with Numbers

In Sects. ?? and **4.4**, this, and in Sects. **8.5**, **10.4**, and **11.4**, we present a number of "standard" examples of operations on logic, sets, numbers, Cartesians, lists and maps. In this section with numbers.

4.4.1 Some Preliminary Remarks

Let m, n be any two, not necessarily distinct natural numbers and i, j any two, not necessarily distinct integers.

value m,n:**Nat**; i,j:**Int**

We assume the defintion of the predecessor and successor functions and 0 (zero()).

4.4.2 Functions and Predicates

4.4.2.1 Simple Number Functions

44. The **abs**olute number, a *natural number*, m, of an *integer*, i,
45. is

44.
45.

46. The **ceil**ing of a real number r, is the integer, i
47. which is r if r is [also] an integer,
48. or else is the integer which is ...

```
46. ceil: Real → Int
46. ceil(r) ≡
47.    if r:Int then r
48.    else ... end
```

49. The **floor** of a real number r, is the integer, n
50. which is r if r is [also] an integer r,
51. or else is the integer which is ...

49. floor: **Real** → **Int**
49. floor(r) ≡
50. **if** r:**Int** **then** r
51. **else** ... **end**

52. To **add**[4] two natural numbers is to

53. ask *is m zero, then the result is n 0 else the result is the same as adding m-1 to n+1, i.e., pre(m) to suc(n)*:

52. add: **Nat**×**Nat** → **Nat**
53. add(m,n) ≡ **if** m=0 **then** n **else** add(pre(m),suc(n)) **end**

54. To **subtract**[5] a number n from a number m is to

55. ask: *is n 0 then the result is m else the result is the same as subtracting n-1 to n-1, i.e., pre(m) from pre(n)*:

54. sub: **Nat**×**Nat** → **Int**
55. sub(m,n) ≡ **if** n=0 **then** m **else** sub(pre(m),pre(n)) **end**

56. To **multiply**[6] two natural numbers m and n is

57. to ask:

    (a) *If m is 0*

    (b) then the result is 0,

    (c) else

    (d) if m is 1

    (e) then the result is n

    (f) else the result is the same as multiplying pre(n) with the multiplication of sum of m with predecessor of n:

[4] usually, in MoLA, we use the operator '+', but shall here "define" that '+'! The MoLA '+' applies to all numbers, not just natural numbers.

[5] usually we use the operator '-', but shall here "define" that '-'! The MoLA '-' applies to all numbers, not just natural numbers.

[6] usually we use the operator '*', but shall here "define" that '*'! The MoLA '*' applies to all numbers, not just natural numbers.

```
56.  mul: Nat×Nat → Nat
57.  mul(m,n) ≡
57a.     if m=0
57b.        then 0
57c.        else
57d.           if m=1
57e.              then n
57f.              else mul(sum(n),pre(n)) end end
```

58. To "lift", **exponentiate**, a natural number m to the n'nth *power*,[7] is

59. to ask:

    (a) *if m is 0*

    (b) then the result is 0,

    (c) else if m is 1

    (d) then the result is m

    (e) else the result is the same as the multiplication of m to m to the power of n-1:

```
58.  exp: Nat×Nat → Nat
59.  exp(m,n) ≡
59a.     if m=0
59b.        then 0
59c.        else if m=1
59d.                then m
59e.                else mul(m,exp(m,pre(n))) end end
```

60. The **mod**ulo of two *natural numbers*, m, n, is a *real* number which

61. is the *r*emainder, a *real number*, of the Euclidean division of m by n, where m is the *dividend* and n is the *divisor*.

```
60.  mod: Nat×Nat → Real
61.  mod(m,n) ≡
61.
```

[7] Usually, in MoLa expressed as m↑n The MoLa '↑' applies to exponents that are, as here, natural numbers, but to *bases* that may be any number, but here we shall define the meaning of that ↑ operator!

4.4.2.2 Number Theory Functions

Example 41 **Fatorials:** ■

Example 42 **Primes:** ■

Example 43 **Greatest Common Denominator:** ■

Example 44 **:** ■

4.4.3 Functions over Booleans, Sets and Numbers

Example 45 **Largest Number in Set of Numbers:**

62.

63.

62.
63.

Example 46 **Sum of Numbers in Sets:**

64.

65.

64.
65.

Example 47 **Partition of Number Sets:**

66.

67.

66.
67.

Example 48 **Partition into Equal Weight Sets:**

68.

69.

68.
69.

4.5 Peano's Axioms

4.5.1 From the Internet

Unabashedly, for the moment, till any possible publication, we "lift" from `https://en.-wikipedia.org/wiki/Peano_axioms`[8]:

The Peano axioms define the arithmetical properties of natural numbers, usually represented as a set N or $\mathbb{N}$. The non-logical symbols for the axioms consist of a constant symbol 0 and a unary function symbol S.

The first axiom states that the constant 0 is a natural number:

1 0 is a natural number.

Peano's original formulation of the axioms used 1 instead of 0 as the "first" natural number, while the axioms in Formulario mathematico include zero.

The next four axioms describe the equality relation. Since they are logically valid in first-order logic with equality, they are not considered to be part of "the Peano axioms" in modern treatments.[7]

2 For every natural number $x, x = x$. That is, equality is reflexive.

3 For all natural numbers x and y, if $x = y$, then $y = x$. That is, equality is symmetric.

4 For all natural numbers x, y and z, if $x = y$ and $y = z$, then $x = z$. That is, equality is transitive.

5 For all a and b, if b is a natural number and $a = b$, then a is also a natural number. That is, the natural numbers are closed under equality.

The remaining axioms define the arithmetical properties of the natural numbers. The naturals are assumed to be closed under a single-valued "successor" function S.

6 For every natural number $n, S(n)$ is a natural number. That is, the natural numbers are closed under S.

7 For all natural numbers m and n, if $S(m) = S(n)$, then $m = n$. That is, S is an injection.

8 For every natural number $n, S(n) = 0$ is false. That is, there is no natural number whose successor is 0.

[8]Giuseppe Peano was an Italian mathematician and glottologist. The author of over 200 books and papers, he was a founder of mathematical logic and set theory, to which he contributed much notation. The standard axiomatization of the natural numbers is named the Peano axioms in his honor. `[Wikipedia]`. Born: August 27, 1858, Cuneo, Italy Died: April 20, 1932, Turin, Italy

The chain of light dominoes, starting with the nearest, can represent N, however, axioms **18** are also satisfied by the set of all light and dark dominoes. The **9**th axiom (induction) limits N to the chain of light pieces (”no junk”) as only light dominoes will fall when the nearest is toppled.

Axioms **1, 6, 7, 8** define a unary representation of the intuitive notion of natural numbers: the number 1 can be defined as $S(0)$, 2 as $S(S(0))$, etc. However, considering the notion of natural numbers as being defined by these axioms, axioms **1, 6, 7, 8** do not imply that the successor function generates all the natural numbers different from 0.

The intuitive notion that each natural number can be obtained by applying successor sufficiently often to zero requires an additional axiom, which is sometimes called the axiom of induction.

9 If K is a set such that:
0 is in K, and
for every natural number n, n being in K implies that $S(n)$ is in K,
then K contains every natural number.

The induction axiom is sometimes stated in the following form:

10 If ϕ is a unary predicate such that:
$\phi(0)$ is true, and
for every natural number $n, \phi(n)$ being true implies that $\phi(S(n))$ is true,
then $\phi(n)$ is true for every natural number n.

In Peano’s original formulation, the induction axiom is a second-order axiom. It is now common to replace this second-order principle with a weaker first-order induction scheme. There are important differences between the second-order and first-order formulations.

4.5.2 A Formal Functional Formulation

Peano’s axioms are reformulated in MoLa !

value
(m,n,x,y,z):(**Nat**×**Nat**×**Nat**)

1. zero: **Unit** → **Nat**
1. zero()

2. eq: **Nat** × NAT → **Bool**
2. eq(n,n) ≡ **true** [eq is *reflexive*]
3. eq(x,y) ⇒ eq(y,x) [eq is *symmetic*]
4. eq(x,y)∧eq(y,z) ⇒ eq(x,z) [eq is *transitive*]
5. ∀ a:**Nat**,b • eq(a,b) ≡ b:**Nat**

is_zero: **Nat** → **Bool**
is_zero(n) ≡ eq(n,zero())

6. suc: **Nat** → **Nat**
6. suc(n):**Nat** [numbers are *closed* under suc]
7. eq(suc(m),suc(n)) ≡ eq(m,n) [suc is an *injection*]
8. ∼eq(suc(n),zero()) [there is no natural number whose successor is zero()]

9. K
9. zero():K
9. ∀ n:**Nat** • n:K⊃suc(n):**Nat**
9. K [contains every natural numbers]

10. ϕ:**Nat**→**Bool**
10. ∀ a:**Nat** • (ϕ(n)⊃ϕ(suc(n)))⊃∀:b:**Nat** • ϕ(b)

4.6 Syntax

By *syntax* we shall mean the arrangement of elements (e.g., words or parts) and their composition (e.g., phrases or composite parts) to create well-formed structure (e.g., sentences or parts) in a language or model. [By words and phrases we mean those of a (written/spoken) laguages; and by parts we mean those of a domain model.]

We present, below, a third version[9] of a syntax, a so-called `BNF Grammar`[10], of numerals whose value are numbers.

BNF Grammar: Numerals and Number Expressions

1.	<Digit>	::=	0|1|2|3|4|5|6|7|8|9
2.	<NatNumeral>	::=	<Digit>
3.		|	<Digit><NatNumeral>
4.	<Numeral>	::=	<NatNumeral>
5.		|	<NatNumeral> **.** <NatNumeral>
6.	<Num-Expr>	::=	<Numeral>
7.		|	<Pre-Num-Expr>
8.		|	<Inf-Num-Expr>
9.	<Pre-Num-Expr>	::=	<Num-Pre-Op><Num-Expr>
10.	<Inf-Num-Expr>	::=	<Num-Expr><Num-Inf-Op><Num-Expr>
11.	<Pre-Num-Op>	::=	−
12.	<Inf-Num-Op>	::=	−|+|∗|/|↑

[9]Sections **2.8** on page 59 and **3.6** on page 73 brought a first and second versions
[10]https://en.wikipedia.org/wiki/Backus-Naur_form

4.7 Closing

4.7.1 Summary

- Number Presentations, Sect. **4.1**, $\boxed{\text{more to come}}$
- Number Types, Sect. **4.2**, $\boxed{\text{more to come}}$
- Operations on Numbers, Sect. **4.3**, $\boxed{\text{more to come}}$

Section **4.4**, Playing Around with Numbers, $\boxed{\text{more to come}}$

- Peano's Axioms, Sect. **4.5**, $\boxed{\text{more to come}}$

4.7.2 Conclusion

This chapter has hinted at the wider, and a more serious, topic of Number Theory. The seminal work on that is:

- Hardy & Wright: Introduction to Number Theory, [49].

4.8 Exercises

Exercise 16 **XNumbers:** *[We refer to Sect.* **??** *on page ??.]*

Exercise 17 **YNumbers:** *[We refer to Sect.* **??** *on page ??.]*

Exercise 18 **ZNumbers:** *[We refer to Sect.* **??** *on page ??.]*

Chapter 5

Names and Values, Characters and Text

Contents

In this chapter we shall introduce the programming cum modelling language MoLa's concepts of **names and values** and **characters and text**.

Motivation: Names & Values

Just as nouns and verbs are indispensable for our everyday use of informal language, so names, in the formal sense of names in any formal specification language, are indispensable. Names serve to identify types and values, whether they are names of functions (corresponding to, say, verbs) or not functions (corresponding to, say, nouns).

Motivation: Characters and Texts, I

Mankind has invented *sign languages*, from about a 100.000 years ago[1] as primitive "scribbling" on cave walls, to *characters* and *texts* the latter composed, in sequences, or as we shall abstract them, as lists, of characters. Most of use learn it, as the second thing we learn[2], from before we go to school! To *read* and to *write* texts.

Motivation: Characters & Text, II

Whatever we write down and later read and edit: it is text composed from characters. Characters and text formed from these characters are elements, therefore, also of the MoLa language. We shall not need to express characters and text till late in this book. But we shall make use of characters and text in examples before that![3] Characters and text directly express their *denotation* [*designation*].

Study Hint: *This chapter does not lend itself to direct "translation" into specific class hour teaching. Rather its "message" should be firmly positioned in teachers' mind – and show up in class as 23-5 minute "detours" when occasioned! This chapter must be studied accordingly: as providing a modicum of knowledge – and thus read in small quanta!*

---

[2]https://bss.au.dk/en/cognition-and-behavior-lab/for-participants/examples-of-studies-in-cobe-lab/why-humans-started-using-symbols

5.1 Names and Values

5.1.1 Identifiers

Definition 61 Identifier: *By an identifier we shall understand a sequence of lower- and uppercase alphabetic characters, say a, b, ..., z, A, B, ..., Z, starting with such an alphabetic character, but possibly interspersed with digits, one or more, 0, 1, ..., 9, and possibly with a properly in-fixed, embedded, single underline character, _, at most one "at a time!"* ■

Example 49 Identifiers: a, aa, abc, a1, a_1 etc. ! ■

5.1.2 Names

As a teaser we start with this •'ed line:

- 7, |||[4], seven[5], 007, sieben[6], sept[7], syv [8]

What do Yo see ? You see seven **names** for one and the same unique mathematical entity. No one has ever "seen" that **value** ! In Sect. **4.5** on page 85 we learned some properties of that value: *that it is the successor of the value named* **6**, *the predecessor of the value named* **8**, etc.

We shall distinguish between

- **value** names: these are names which denote, i.e., refer to values: Booleans, sets, numbers, Cartesians, lists, maps, functions, i.e., any kind but types.
- **type** names: these are names which denote, i.e., refer to types, i.e., special classes of values.
- **variable** names: these are names which denote, i.e., refer to so-called "assignable", declared MoLa variables.
- **channel** names: these are names which denote channels, i.e., means of communication between MoLa behaviours.

5.1.3 Specific Non-function Names

5.1.3.1 Booleans

In Sect. **2.2.1** on page 51 we introduced the two Booleans [bold-faced] identifiers (they are literals): **true** and **false** as names for respective mathematical truth values.

[4] ||| is a binary, i.e., a radix 2, name
[5] English
[6] German
[7] French
[8] Danish

5.1.3.2 Numerals

Numerals were introduced in Sect. **4.1** on page 78.

We repeat: we refer to identifiers, i.e., names of numbers as *numerals*.

So, please, from now on, be careful when You refer to numbers, whether You actually refer to [speak of] numbers, or to [of] their names: numerals!

Definition 62 Numerals: *Numerals come in different "shapes", each according to whether you wish to express a natural number or a rational number.*

The **syntax** *of natural number numerals is that of a sequence of one or more* *digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.*

For **example:** *123456789!*

The **syntax** *of rational, but not necessarily natural, number numerals is that of a sequence of two sequences, first one or more digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, then a period, ".", and then another sequence of digits.*

For **example:** *1234.56789* ■

5.1.3.3 Value and Types

By a *value name* we shall understand an identifier which denotes, i.e., stands for a *value* of some *type*.

We, i.e., You (!), introduce value names by a conscious effort. You choose to "set aside" some value, one that we/You, later in our/Your development of an MoLa specification, a model, or a program, wish to return to – by referring to it. You do so by writing either of the the following in MoLa:

either:	**or:**[9]
value v:T	**value** v:T = $\mathcal{E}$(...)

where T is a type [identifier, i.e., the name of a type – one that we/You have already introduced (i.e., defined), or one that we/You later intend to introduce; and where $\mathcal{E}$(...) is an expression which supposedly evaluates to a value of type T.

In the **either** alternative, the value name v is being expressed as *"any value of type T"*.

In the **or** alternative, the value name v is being expressed as *"the specific value $\mathcal{E}$(...) of type T"*.

5.1.3.4 Variable Names

Although we shall not, for a long "time", treat the concept of *imperative programming* we shall already here outline the most basic concepts of imperative programming, namely those of *variables*, their *declaration* and *assignment*. The reader may therefore skip this section till reaching Chapter **17**.

[9] The **either:** and **or:** is not an element of MoLa!

Definition 63 Variable: *By a* ***variable*** *we shall here understand a "placeholder", i.e., a place ["somewhere"] that hold ["keep, store"] values where these placeholder values may change, by so-called* ***assignment*** ■

Definition 64 Variable Declaration: *By a* MoLA ***variable declaration*** *we shall here mean a* MoLA ***specification unit*** *which introduces a* ***variable name****, its* ***type*** *and, usually,* ***initial value*** ■

as exemplified by:

variable v:T := expr

We say that variable v is *declared* .

Definition 65 State, I: *By a* ***state*** *we shall here mean the "sum total" set of values kept by, i.e., stored in, a* MoLA *specification's declared variables* ■

Definition 66 Assignment: *By* ***assignment*** *we shall here understand an* ***action*** *which changes a state, by replacing its "current" value with that of the assignment* ■

v := expr ;

The above exemplifies a MoLA *assignment*. The above clause – v := expr ; – is to be thought of as being interpreted, i.e., evaluated, at some time during the evaluation of the MoLA specification in which it occurs. "Immediately" before that interpretation the value of variable v may be one. Immediately "after" the interpretation the value of variable v will be that of the value of expression expr.

5.1.3.5 Channel Names

Although we shall not, for a long "time", treat the concept of *parallel programming* we shall already here outline two of the most basic concepts of parallel programming, namely those of *behaviour*, *declaration* of *channels* and *communication* – between behaviours – over *channels*. The reader may therefore skip this section till reaching Chapter **18**.

By a MoLA *behaviour* we shall here understand the time-ordered sequence of evaluation of MoLA clauses: of value and variable names, of simple and composite expressions, of statements of imperative clauses, if any, and of communication clauses, see below, if any ■

By a *channel* we shall here mean a means for communication: synchronisation and exchange of values between behaviours ■

By a MoLA *channel declaration* we shall here mean a MoLA *specification unit* which introduces a *channel name* and the *type* of its messages ■
Thus:

channel ch:M

is an example MoLA specification unit. It introduces the channel name ch and the type, M of the messages that may be communicated over the channel.

Definition 67 Channel Communication: *By channel communication we shall here mean the expression of the offer, !, of one behaviour, β_o, to "output" a value to another behaviour: ch ! value with the expectation of the offering behaviour, β_o, that there is another behaviour, β_i which is willing to input, i.e., to accept, ?, that communication: ch ?* ■

That is:

β_o: ch ! value

β_i: ch ?

are clauses in respective MoLA specifications of behaviours β_i and β_o.

5.1.4 Function Names

5.1.4.1 Operator Names

5.1.4.1.1 Boolean Operator Names Section **2.4** on page 52 introduced the Boolean operators[10]:

- $\sim, \wedge, \vee, \supset$, and $=$.

5.1.4.1.2 Set Operator Names Section **3.4.1** on page 68 introduced the set operators

- $\cup, \cap, \in, \setminus, \subset, \subseteq, \notin$ and **card**.

5.1.4.1.3 Arithmetic Operator Names Section **4.3.1** on page 79 introduced the arithmetic operators

- $+, -, *, /, =, \neq, <, \leq, >$, and $\geq$.

5.1.4.1.4 Cartesian Operator Names Section **8.4** on page 113 introduced the Cartesian decomposition clause:

- **let** (a,b,...,c) = $\mathcal{C}$ **in** ... **end**

We may consider that a *"distributed-fix"* operator which applies to Cartesian-valued expressions, $\mathcal{C}$ by ascribing user chosen names, i.e., the identifiers a, b, ..., c to its components[11].

The '...' between **in** and **end** above is usually some clause of the general form $\mathcal{E}$(a,b,...,c).

[10] The • is not an operator name!

[11] For more on this *"pattern-driven"* decomposition we refer to Sect. **16.1** on page 186.

5.1.4.1.5 List Operator Names Section **10.3** on page 137 introduced the list operators

- **hd, tl, card, elems, inds**, $\widehat{}$, $=$, and $\neq$.

5.1.4.1.6 Map Operator Names Section **11.3** on page 147 introduced the map operators:

- $\cdot(\cdot)$, **dom, rng**, $\dagger$, $\cup$, $\backslash$, $/$, $=$, $\neq$, $\equiv$ and $^\circ$.

5.1.4.2 Function Names and Types

We refer to Chapter **6** for coverage of functions, function names and values.

5.1.5 Tokens and Parts

In[12] Chapter **1** Sects. **1.6.1** on page 30 and **1.6.2** on page 36 we dealt with the external qualities of endurant parts and their internal qualities. To remind the reader: the internal qualities are those of *unique identifiers* Sect. **1.6.2.1**, *mereologies* Sect. **1.6.2.2** and *attributes* Sect. **1.6.2.3**.

In this section we shall elaborate a bit more on the domain concept of *parts*, and on a concept of *tokens*.

Definition 68 Token: *By a token we shall understand a quantity which is of some token type, is usually referred to by a token name, and has a token value* ■

Definition 69 Token Type: *By a token type we shall understand a type, that is, a class of token values* ■

Definition 70 Token Name: *By a token name we shall understand an identifier* ■

Definition 71 Token Value: *By a token value we shall understand a quantity which we can think of as atomic, that is, of no further structure, and which we need not be concerned with* ■

With *parts*, whether *atomic* or *composite*, we shall associate tokens. Their unique identifiers, cf. Sect. **1.6.2.1** on page 36, are tokens. We may choose to abstract one or more *part attributes* in the form of tokens.

Definition 72 Unique Identifier Types and Observers: *Let U be any part type and let UI be the type of its unique identifiers; then the formalization:*

[12] **Change:** *New text inserted: Tokens and Parts*

type
U, UI
value
uid_U: U → UI ■

formalizes how we express introduction and use of (i.e., access to) unique identifiers of parts in MoLa. The boldfaced **uid_** prefix to a part name shall indicate that **uid_**U is a meta-function, that is, one that is not describable, but "just does the trick!".

Definition 73 Attribute Types and Observers: *Let U be any part type and let A be the type of one of its attributes, possibly of token type, then the formalization:*

type
A
value
attr_A: U → A ■

formalizes how we express introduction to and us of (i.e., access to) attributes of parts in MoLa. The boldfaced **attr_** prefix to a part name shall indicate that **attr_**A is a meta-function, that is, one that is not describable, but "just does the trick!".

Example 50 Unique Identifier Tokens: With hubs, links and automobiles of a road transport we associate unique identifiers:

type
HI, LI, AI
value
uid_H: H → HI
uid_L: L → LI
uid_A: A → AI ■

Example 51 Attributes Tokens: With hubs, links and automobiles of a road transport we associate the respective token attributes.

70. Hubs have street intersection names, HN.

71. Links have street names, LN.

72. Automobiles have makers names (Ford, GM, Tesla), AM.

type
70. HN
71. LN
72. AM

value

70. **attr**_HN: H → HN
71. **attr**_LN: L → LN
72. **attr**_AM: A → AM ■

5.2 Characters and Texts

to come

5.2.1 Signs

There is an auxiliary notion of ***signs***. Signs are not “free-standing” MoLa values. They occur in the context of characters and texts. Here are some example of signs that may occur as part of MoLa characters, or in MoLa texts.

> **a**, **b**, ..., **z**, **A**, **B**, ... **Z**, **0**, **1**, ..., **9**, **.**, **,** , **.,** , **;** , **.;** , **:** , **.** , etc.

as per Your choice! Notice, in the last seven explicitly shown signs, the space, and the space after the punctuation (etc.) marks as an integral element of the sign,

5.2.2 Characters

to come

5.2.2.1 Character Literals

We shall represent characters in single quoted signs: Example of characters are

> **'a'**, **'b'**, ..., **'z'**, **'A'**, **'B'**, ..., **'Z'**, **'0'**, **'1'**, ..., **'9'**, **'.' ' '**, **', ' '.,** ',, **'; '**, **'.; '**, **': '**, **'. '**, etc.

5.2.2.2 Character Type

We name the class of all characters **Char**. **Char** is a type name.

> **Char**

It is part of MoLa. You do not have to *introduce* that type. [Since it is directly provided by MoLa, “built-in” .]

5.2.2.3 Character Operations

to come

> $=, \neq$

5.2.3 Texts

5.2.3.1 Text Type

to come

Type

to come

5.2.3.2 Text as Character Strings

Texts in MoLa are sequences of signs enclosed in double quotes: ". An example:

- **"Dines Bjørner, Born 4.10.1937"**

5.2.3.3 Text Operations

Texts otherwise are like lists, see Chapter **10**, That is, the list operators apply to texts: You can perform the following operations on texts:

- the first, the **h**e**a**d, character of a text: **hd** text [**'B'**];
- the text of all but the first character, i.e., the **t**ai**l**, of a text: **tl** text [**"ines Bjørner, Born 4.10.1937"**];
- the **set** of all distinct characters of a text: **elems** text [{**'D'**, **'i'**, **'n'**, **'e'**, **'s'**, **' '**, **'B'**, **'j'**, **'ø'**, **'r'**, **','**, **'o'**, **'4'**, **'.'**, **'1'**, **'0'**, **'9'**, **'3'**, **'7'**}];
- the set of all **indices** of the characters of a text: **inds** text [{1,..,28}];
- texts can be **concatenated**: text_1^text_2^...^text_n; and
- characters can be **selected** from a text: text[i] [**"Dines Bjørner, Born 4.10.1837"**[9]=**ø**].

to come

...

to come

5.3 Closing

5.3.1 Summary

to come

5.3.2 Conclusion

$\boxed{\text{to come}}$

5.4 Exercises

Exercise 19 **XNamValChaTxt:**

Exercise 20 **YNamValChaTxt:**

Exercise 21 **ZNamValChaTxt:**

Chapter 6

Functions

Contents

Motivation: Functions

All our actions, in whichever domain we are present, are functions. Nothing "functions" without functions. The things, the entities, observe in the world around us, exhibit two sides: the stable, static, enduring, lasting kind that let's us observe them, and the dynamic side in which these entities transcendentally "morph" into functions: behaviours, actions and events that occur over times.

Study Hint: *This is a short chapter. It relies on our having introduced the concept of functions already from Chapter* **2** *and in all subsequent chapters till now. So the chapter summarizes what we have up till now. Read it in one go!*

6.1 Function Definitions

It is a bit too early, in this book, to detail the MoLa concept of function definitions, but there has to have been already a few examples. So we attempt the follow.

Functions can be defined in a number of ways.

6.1.1 Simple Functions

6.1.1.1 Direct Definition

A *direct function definition* has the form:

[1.] **value**
[2.] f: A → B
[3.] f(a) ≡ $\mathcal{E}$(a)

- The above, [1.-3.] is a MoLa *specification unit*.
- Line [1.] signals, to the reader, that a value, f, is being introduced or defined;
- Line [2.] defines the so-called *signature* of function f: that its name is f, that the type of its formal argument is A, that the value is a function, →, and that the type of its result is B – where A and B are types names introduced or defined elsewhere;
- Line [3.] then presents the formal invocation, f(a), of the function and it "body" of definition: $\mathcal{E}$(a) – where $\mathcal{E}$(a) is any MoLa expression in which a, the *formal argument*, occurs free, but here being bound by the a in the left-hand side f(a).

Example 52 **Some Simple Functions:** We "immodestly" assume the natural [**Nat**], integer [**Int**], and real [**Real**] number types and the arithmetic operators of +, -, * and /:

value
  sum: **Nat** × **Nat** → **Nat**
  sum(a,b) ≡ a+b
  min: **Nat** × **Nat** → **Nat**
  min(a,b) ≡ a−b
  mpy: **Nat** × **Nat** → **Nat**
  mpy(a,b) ≡ a∗b
  div: **Nat** × **Nat** $\stackrel{\sim}{\to}$ **Int**
  div(a,b) ≡ a/b **pre**: b≠0

■

Please accept that we are somehow "cheating" in that we use the conventional +, −, ∗, / operators to explain sum, min, mpy and div – somewhat tautologically! The idea was to just show how ordinary function definition [specification units] look like.

6.1.1.2 Indirect Definition

An *indirect function definition* has the form:

[1.] **value**
[2.] f: A $\rightarrow$ B
[3.] f(a) **as** b
[4.] **pre**: **as** $\mathcal{P}$(a,b)

- The above, [1.-4.] is a MoLa *specification unit*.
- Line [1.-2.] is as before;
- Line [3.] expresses that the result of f(a) is b
- Line [4.] where a,b satisfies a predicate $\mathcal{P}$(a,b).

Example 53 xxx:

xxx

■

6.1.2 Partial Functions

The last example above: the division function, illustrated the concept of partial function, a function that is not defined for some of its arguments – of the type [otherwise] specified. For such functions we show, in the function signature, that the function is not total, by $\rightarrow$, and by the **pre**-condition following the function definition body, if possible, the logical condition for which values of the argument[s] it may fail to evaluate:

type
  A, B
value
  f: A $\xrightarrow{\sim}$ B
  f(a) $\equiv$ $\mathcal{E}$(a) **pre**: $\mathcal{P}$(a)

where $\mathcal{P}$(a,b) is some suitable predicate.

6.1.3 Recursive Functions

Recursive functions can be defined:

[1.] **value**
[2.] f: A → B
[3.] f(a) ≡ $\mathcal{E}$(f,a)

Here, in [3.] f may occur [*free*] in $\mathcal{E}$(f,a) but being ***bound*** by the ≡ to the left-hand occurrence of f in f(a).

Example 54 **A Recursive Function Definition:**

value
fact: **Nat** → **Nat**
fact(n) ≡ **if** n=0 **then** 1 **else** n∗f(n−1) **end** ∎

6.2 Function Definition and Range Sets

Functions accept argument of their ***definition set*** and yield results in their ***range set***.

Definition 74 **Definition Set:** *By the definition set of a function (or a map) is meant the set or argument values for which the function (or map) is well-defined* ∎

Given a defined function, f, we cannot, in general determine its definition set.

Definition 75 **Range Set:** *By the range set of a function (or a map) is meant the set or result values yielded by the function (or map) when it is applied to argument values of its definition set* ∎

Given a defined function, f, **one cannot**, in general determine its range set. The reason for this is that it is in general undecidable whether evaluation of a function, from its non-map definition, will terminate[1]. For maps **one can** determine these sets, see Sect. **11.3** on page 147.

6.3 Summary and Conclusion

6.3.1 Summary

$\boxed{\text{to be written}}$

[1]**Editorial note:** More on this.

6.3.2 Conclusion

$\boxed{\text{to be written}}$

6.4 Exercises

Exercise 22 **XFctTyp:**

Exercise 23 **YFctTyp:**

Exercise 24 **ZFctTyp:**

Chapter 7

Infinity

Contents

In this chapter we shall elaborate on the concept of **infinity**.

Infinity is that which is boundless, endless, or larger than any natural number. It is often denoted by the infinity symbol ∞. Infinity is not a number.

If, wrt. numbers we wish to indicate "the cardinality of an infinite set or list" we use the Hebrew symbol $\aleph_0$, pronounced Aleph 0.

We "borrow" from a number of sources: `https://www.britannica.com/science/infinity-mathematics`, ... $\boxed{\text{MORE TO COME}}$

7.1 Motivation

```
Chapter on Infinity
D = ...  D  ...
D = D-set
D = D*
D = D -m-> D
D = D -> D, D = D -~-> D
```

7.2 Treatment of Infinity in Terms of Numbers

7.3 Cardinality of Sets

7.4 Countably Infinite Sets

7.5 Closing

7.5.1 Summary

7.5.2 Conclusion

7.6 Exercises

Exercise 25 XInfinity:

Exercise 26 YInfinity:

Exercise 27 ZInfinity:

Chapter 8

Cartesians

Contents

In this chapter we shall introduce the programming cum modelling language MoLa's concept of **Cartesians**. The French mathematician Réne Descartes (1596–1650) introduced the concept that we shall call *Cartesians*.

Motivation: Cartesians

Humans tend to group, "to pair", to put in "triplets", or "quadruplets", etc., domain phenomena we observe, concepts we perceive, etc. In domain phenomena this seems unavoidable: that endurants, parts, are composed from a definite occurrences of sub-parts, etc.

8.1 Informal Presentations of Cartesians

A Danish Drivers License presents the following *fields* of information[1]:

- LN (last name) [1.],
- FN (first name) [2.],
- BD (birth date and country) [3.],
  - ID (issue date) [4a.],
  - ED (expiry date) [4b.],
  - IA (issuing authority) [4c.],
  - NI (national identity number) [4d.],
- LI (license id.) [5.],
- S (signature) [7.],
- P (photo) [8.], and C (code) [9.].

By LN, FN, BD, ID, ED, IA, NI, LI, S, P and C we shall here mean both the name of these fields and their 'data'. An example driver's license could then be: (Bjørner,Dines,4 Oct. 1937,...,...,Rigspolitiet,...,...,...)[2]

We can represent these fields of information as the Cartesian:

- (T,C,LN,FN,NId,ND,G,PN,,ID,ED,I,NI,S,P).

8.2 Formal Presentation of Cartesians

to be written

8.2.1 Formal Presentation of Cartesian Types

In mathematics, the *Cartesian*[3] *product* of sets s and s' is defined as the set of all ordered pairs (x, y) such that x belongs to s and y belongs to s'.

[1] See photo in Fig. **8**.2 on page 116

[2] – where we have left out more-or-less confidential information, see, Fig. **8**.2 on page 116!.

[3] René Descartes (born March 31, 1596, Descartes, France, died February 11, 1650, Stockholm, Sweden) was a French philosopher, scientist, and mathematician, widely considered a seminal figure in the emergence of modern philosophy and science. Mathematics was central to his method of inquiry, and he connected the previously separate fields of geometry and algebra into analytic geometry. `[Wikipedia]`

- $s \times s' \equiv \{(x,y)|x \in s \wedge y \in s'\}$

In other words: we use the symbols (and) to delimit, to enclose, a ***grouping***, $(x, y, ..., z)$, of two or more entities, $x, y, ..., z$, to denote a Cartesian.

We justify the introduction of Cartesians as follows:

$\boxed{\text{more to come}}$

8.2.2 Simple Cartesian Type Definitions

Let A, B, ..., C be types, i.e., identifiers denoting types. Then:

- **type** $K = A \times\ B \times\ ... \times\ C$

defines K to be a type whose values are Cartesians over elements of types A, B, ..., C, in that order:

K = {| (a,b,...,c) | a:A, b:B, ..., c:C |}

Here {| and |} are type-forming — rather than set-forming — delimiters.

8.2.3 Named Cartesian Type Definitions

Let mkK be an identifier, then

- **type** $nK :: A \times\ B \times\ ... \times\ C$

defines a type of marked (i.e., named) Cartesians:

nK = {| mkK(a,b,...,c) | a:A, b:B, ..., c:C |}

That is: the nK :: ... induces the identifier $mknK$; the nK is taken from the identifier to the immediate left of ::; the mk stands for *make*. The specifier, i.e., You!, must take care to use the text nK :: ... at most once in a full specification.

8.2.4 Cartesian Element Selectors

For Cartesians, whether simple or named, we can further introduced so-called [Cartesian element] ***selectors***. Cartesian element ***selectors*** are simple functions which apply to Cartesians and yield named elements of these. Their definition is as follows:

- **type** $K = (s\_a$:A×s_b:B×...×s_c:C)

- **type** K' :: $(s\_a$:A×s_b:B×...×s_c:C)

where $s\_a, s\_b, ..., s\_c$ are distinct, are referred to as ***selectors***, and a chosen by You!

Their "meaning" is as follows: Let k be some Cartesian, say $(alpha, beta, ..., gamma)$. Then $s\_a(k) = alpha, s\_b(k) = beta, ..., s\_c(k) = gamma$.

8.3 Cartesian Expressions

8.3.1 Simple Cartesian Expressions

Given values (say, in the form of variable names) $a, b, ..., c$ of respective types $A, B, ..., C$, then

- $(a, b, ..., c)$

is an expression which evaluates to a Cartesian value in $(A, B, ..., C)$, i.e., in K. Likewise

- $mkK(a, b, ..., c)$

is an expression which evaluates to a Cartesian value in $mkK(A, B, ..., C)$, i.e., in nK.

8.3.2 "let ... in ... end" Cartesian Expressions

The clause

- **let** $identifier\_pattern \; = \; \mathcal{E}_d(...)$ **in** ... **end**

is said to *decompose* the value of expression $\mathcal{E}_{d(...)}$ into constituent values. Let the value of expression $\mathcal{E}_{d(...)}$ be a Cartesian in K, then

- **let** (a,b,...,c) $= \mathcal{E}_{d(...)}$ **in** $\mathcal{E}(a, b, ..., c)$ **end**

identifies the elements of $\mathcal{E}_d(...)$. Similarly Let the value of expression $\mathcal{E}'_d(...)$ be a Cartesian in mkK, then

- **let** mkK(a,b,...,c) $= \mathcal{E}'_d(...)$ **in** $\mathcal{E}$(a,b,...,c) **end**

identifies the elements of $\mathcal{E}'_d(...)$.

8.3.3 Simple Example of Cartesians

Let Last$N$ame, First$N$ame, Birth$D$ate, Issue$D$ate, Expiry$D$ate, Issuing$A$uthority, NatlIdNumber, License Id., Signature and Photo be the fields, i.e., elements of a, in this case, Danish Drivers$L$icense. Then

- $DL :: LN{\times}FN{\times}BD{\times}ID{\times}ED{\times}IA{\times}NId{\times}LI{\times}S{\times}P$

is an example of a Cartesian type which models a [Danish] drivers license. [We omit details of the types $LN, FN, BD, ID, ED, IA, NId, LI, S$ and P.]

8.4 Operations on Cartesians

Besides the **let ... =** $\mathcal{E}$ **in ... end** construction applied to Cartesian $\mathcal{E}$ values there is basically only the relational operator = that can be applied to Cartesians:

- $(a_1, b_2, ..., c_m) = (\alpha_1, \beta_2, ..., \gamma_n)$

holds if the number of elements in the two Cartesians is the same, i.e., $m = n$, for each element $a_1 = \alpha_1, b_2 = \beta_2, ..., c_m = \gamma_n$, and $(a_1, b_2, ..., c_m)$ and $(\alpha_1, \beta_2, ..., \gamma_n)$ are of the same type.

8.5 Playing Around with Cartesians

8.5.1 Some Preliminary Remarks

Let a, a_i, etcetera be any not necessarily distinct values of some type(s).

value a:A, ..., a_i:A_i, ...

We assume the definition of ...

8.5.2 Functions and Predicates

8.5.2.1 Modeling Trees

In this section, and in Sects. **8.5.2.1 10.4.2.1** we apply the set, Cartesian, list and map type concepts to the abstract Modeling of some form of trees.

Figure **8**.1 on the next page informally illustrates an abstract concept of trees. Lines between dots, •, denotes trunks. The bottom dot denote a root. The dots between lines denote branchings, and the uppermost dots denote leaves.

Dots and trunks are distinctly labeled. Dots in sans serif font; trunks in *slanted* font.

Example 55 A Cartesian Tree Type:

73. A Cartesian tree, $\mathrm{T}_{\mathcal{C}}$, has a root, a [main] trunk, a branching, ***and***[4] a set of two or more Cartesian sub-trees.

74. A root is presently modeled by a [further unspecified] root identifier.

75. A trunk is presently modeled by a name, i.e., a [further unspecified] trunk identifier.

76. A branching is presently modeled by a name, i.e., a [further unspecified] branch identifier.

[4]The textual enumeration, separated by commas and ending with ***and*** informs us that a Cartesian, ×, is being defined.

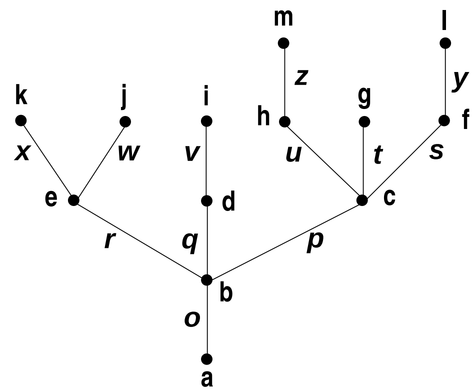


Figure 8.1: An Abstract Node and Vertex Labeled Tree

77. A Cartesian sub-tree, $\mathrm{ST}_{\mathcal{C}}$, is ***either*** a leaf ***or***[5] is a *proper* Cartesian sub-tree.
78. A leaf is presently modeled by a name, i.e., a [further unspecified] leaf identifier.
79. A proper Cartesian sub-tree, $\mathrm{PT}_{\mathcal{C}}$, has a trunk, a branching ***and*** a set of two or more proper Cartesian sub-trees.
80. Root, branch, trunk and leaf identifiers are all [further undefined] quantities of the same "kind", i.e., sort[6].

type
73. $\mathsf{T}_{\mathcal{C}} = \mathsf{RT} \times \mathsf{TR} \times \mathsf{BR} \times \mathsf{ST}_{\mathcal{C}}$**-set**
74. RT = RID
75. TR = TID
76. BR = BID
77. $\mathsf{ST}_{\mathcal{C}} = \mathsf{LF} \mid \mathsf{PT}_{\mathcal{C}}$
78. LF = LID
79. $\mathsf{PT}_{\mathcal{C}} = \mathsf{TR} \times \mathsf{BR} \times \mathsf{PT}_{\mathcal{C}}$**-set**
79. ID = RID | RID | BID | LID
axiom
73. $\forall$ (_,_,_,sts):$\mathsf{T}_{\mathcal{C}}$• **card** sts $\geq$ 2
79. $\forall$ (_,_,pts):$\mathsf{PT}_{\mathcal{C}}$• **card** pts $\geq$ 2 ■

Example 56 Functions over Cartesian Tree Type:

81. (a)
    (b)

[5]The textual ***either or*** informs us that a union type, |, is being defined.
[6]'sort' is just another name for 'type'.

(c)
(d)

82. (a)
(b)
(c)
(d)

83. (a)
(b)
(c)
(d)

84. (a)
(b)
(c)
(d)

type
81.
value
81a.
81a.
81b.
81c.
81d.

8.5.2.2 Functions over Cartesians

85.

86.

85.
86.

87.

88.

87.
88.

8.5.2.3 SQL: Towards a Database Query Language

8.5.2.3.1 A Simple Beginning

Example 57 A Single Relation Database:

89. We equip the definition of the drivers license Cartesian with selectors.
90. Let DB be the set of all drivers licenses of a country, say Denmark.
91. Let Sel be either of the selectors of DL.
92. Let UnaryQuery be ...
93. Let query be a function with two formal arguments: a unary query mkUnaryQuery(uq) and the simple drivers license database db – with that function yielding a set of elements (of the same type) of drivers licenses.
94. The drivers license database db must satisfy the following unique identity criterion: there must be a selector, s_ ...

Figure 8.2: Author's Danish Drivers License

type
89. DL = (s_LN:LN×s_FN:FN×s_BD:BD×s_ID:ID×s_ED:ED×s_IA:IA×s_NId×NId×s_S:S×S_P:P)
90. DB = DL-**set**
type
91. Sel = {|s_LN|s_FN|s_BD|s_ID|s_ED|s_IAs_NId|s_S|s_P|}

92. UnaryQuery :: Sel

value

93. query: UnaryQuery $\times$ DB $\rightarrow$ LN-**set**|FN-**set**|BD-**set**|ID-**set**|ED-**set**|IA-**set**|NId-**set**|S-**set**|P-**set**

93. query(mkUnaryQuery(uq),db) $\equiv$ { uq(k) | k:K • k $\in$ db }

axiom

94. $\forall$ db:DB,$\exists$ uq:Sel • uq=s_ID$\wedge$**card** query(mkUnaryQuery(uq),db) = 1

8.5.2.3.2 A Realistic Database

Example 58 **SQL:**

to be written

8.6 Syntax

By *syntax* we shall mean the arrangement of elements (e.g., words or parts) and their composition (e.g., phrases or composite parts) to create well-formed structure (e.g., sentences or parts) in a language or model. [By words and phrases we mean those of a (written/spoken) laguages; and by parts we mean those of a domain model.]

to be written

8.7 Summary and Conclusion

8.7.1 Summary

to be written

8.7.2 Conclusion

to be written

8.8 Exercises

Exercise 28 **XCartesians:**

Exercise 29 **YCartesians:**

Exercise 30 **ZCartesians:**

Chapter 9

Graphs

Contents

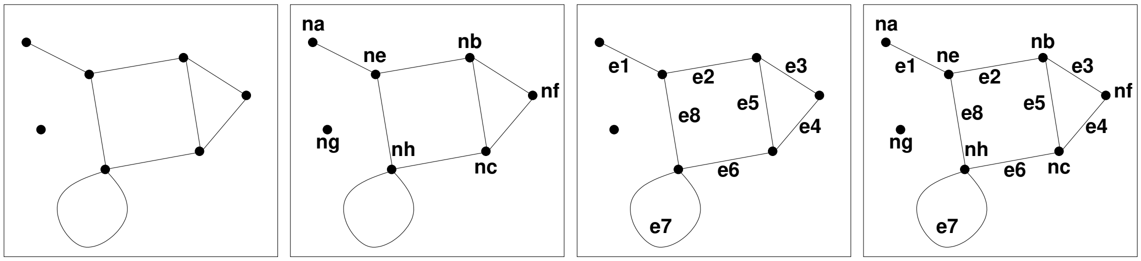


Figure 9.1: Undirected Graphs: Un-labeled, Node-labeled, Edge-labeled, Fully labeled

In this chapter we shall elaborate on the mathematical concept of **graphs**.

Motivation: Graphs

Domain phenomena, such as road, bus, rail, airline and shipping routes are, for human convenience visually, or otherwise abstracted, as graphs.

A special sub-class of graphs are so-called ***trees***. They will be dealt with in Sect. **9.2**.

9.1 Graphs

Definition 76 Graphs: *By a **graph** we shall here mean a collection of vertices and edges that join pairs of vertices – or "loops around" a single node* ■

For mathematical introductions to graph theory we refer to [5, 52].

9.1.1 Informal Presentation of Graphs

9.1.1.1 Undirected Graphs

Examples of graphs are:

Figure **9**.1 shows four graphs. They are all made up from 7 nodes and 8 edges. The leftmost is unlabeled. Number two from left has [only] its nodes distinctly labeled. Number three from left has [only] it edges distinctly labeled. The rightmost has all nodes and edges distinctly labeled. We do not consider the cases where either nodes, or edges, or all may have multiply named labels.

The idea of the edges is to express that the pairs of nodes they connect are (or the single node they "loop around" is) [somehow] "related". [How they are related is not expressed.]

Edges form paths of length one. Two nodes are said to be immediately connected if there is an edge between them. In general a ***path*** of a graph is a sequence of one or more ***adjacent edges*** where adjacency means that the two edges share [at least] one node.

A purpose of labeling is to make it easier for You and me to discuss the graph, to refer to for example, paths, such as the path of number two graph from the left of Fig. **9**.1 on the preceding page, say the one listing the nodes na, ne, nb, nc, nh.

9.1.1.2 Directed Graphs

Figure **9**.2 shows four graphs. Again made up from nodes and edges as in Fig. **9**.1. Now the edges are *directed*, i.e., have arrows, some in one direction only, some in both! Again the graph nodes and edges are labeled as in Fig. **9**.1 on the preceding page.

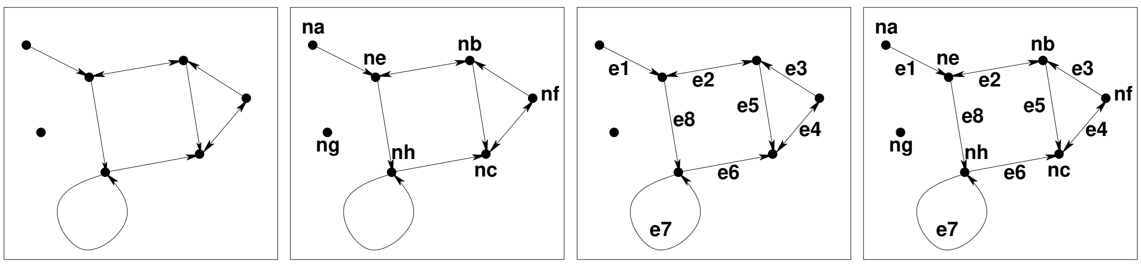


Figure 9.2: Directed Graphs: Un-labeled, Node-labeled, Edge-labeled, Fully-labeled

A purpose of having directed graphs is to "narrow" the range of possible paths of a graph. A edge which is directed from a node, say n to a node, say n', defines a path (n, n'). Thus the sequence na, ne, nb, nc, nh is no longer an admissible path of the second graph from the left of Fig. **9**.2.

9.1.2 Formal Representation of Graphs

We shall consider only graphs whose nodes, and possibly also edges, are labeled.

By a formal presentation of graphs we mean one which relies on such mathematical concepts as sets, Cartesians and, possibly, functions.

9.1.2.1 Set and Cartesian Representation of Graphs

Let V be a type which stands for node, i.e., *vertex*, labels and E be a type which stands for edge labels. Then we can present a class, a type, of all vertex and edge labeled directed graphs as follows.

95. There is a type V of vertex labels.

96. There is a type E of edge labels.

97. A graph is then formalisable as a triplet of vertex labels, edge labels and directed edge prescriptions, where a directed edge prescription is a triplet of a from vertex label, an edge label and a to vertex label.

98. The labels mentioned in directed edge prescriptions must be of the graph,
99. and each edge label mentioned in the set of a graph's edge labels must be of exactly one directed edge prescription.

type
95. V
96. E
97. G = V-**set** × E-**set** × (V×E×V)-**set**
axiom
98. ∀ (vs,es,edges):G • ∀ (vf,e,vt):(V×E×V) • {vf,vt} ⊆ vs ∧ e ∈ es
99. ∀ e:E • e ∈ es ⇒ ∃ (vf,e′,vt):(V×E×V) • (vf,e′,vt) ∈ edges ∧ e=e′

The rightmost graph of Fig. **9**.2 on the previous page is thus presentable as a value in G:

({na,nb,nc,nd,ne,nf,nh},
{e1,e2,e3,e4,e5,e6,e7,e8},
{(na,e1,ne),(ne,e2,nb),(nb,e2,ne),(nf,e3,nb),(nf,e4,nc),
(nc,e4,nf),(nb,e5,nc),(nh,e6,nc),(nh,e7,nh),(ne,v8,nh)})

The above was an example of the classical graph theory presentation of graphs. In our rendition we leave out the well-formedness of the (V×E×V)-**set** wrt. the V-**set** × E-**set** components of G – but see next!

9.1.2.2 A Typed Axiomatic Domain Model of Graphs

We present a *domain model* of graphs. That is, a model where we do not explicitly model in terms of Cartesians, but only of *abstract types* wrt. graphs (G), nodes (N) and edges (E), and *observer functions* (**obs_**, **uid_**, **mereo_** and **attr_**) wrt. how nodes and edges are observed from graphs, G, and connected,

100. Graphs, G, are to be understood as follows:
    (a) From any graph one can **obs_**erve a set, SoN, of nodes, N, and a set, SoE, of edges, E.
    (b) From any node and any edge one can observe their unique identifiers (NI, respectively EI), these are tokens
    (c) [“miraculously”] obtained by means of the unique identifier observer **uid_**
    (d) and no two nodes or edges of a graph have the same unique identifier –
    (e) From any node and any edge one can observe their mereologies, i.e., how they are connected to respectively edges and nodes –
    (f) where the mereology of a node, respectively an edge, is [“miraculously”] obtained by means of the mereology observer **mereo_**,

(g) and where the mereology of a node is a set of 1, or more edge identifiers, and the mereology of an edge is a set of 1 or two edge –

(h) and all are identifiers of the graph.

type
100. G
100a. SoN = N-**set**, N, SoE = E-**set**, E
100b. NI, EI
100f. NMer = EI-**set**, EMer = NI-**set**
value
100. g:G
100a. **obs**_SoN: G → SoN, **obs**_SoE: G → SoE
100c. **uid**_N: N → NI, **uid**_E: E → EI
100d. graph_NIs: G → { **uid**_N(n) | n:N•n ∈ **obs**_N(g) }
100d. graph_EIs: G → { **uid**_E(e) | e:E•e ∈ **obs**_E(g) }
100f. **mereo**_N: N → NMer, **mereo**_E: E → EMer
axiom
100d. graph_NIs(g) ∩ graph_EIs(g) = {}
100d. ∧ **card** graph_NIs(g)+**card** graph_EIs(g) = **card obs**_SoN(g) + **card obs**_SoE(g)
100g. ∧ ∀ n:N,e:E • n ∈ obsSoN(g) ∧ e ∈ **obs**_SoE(g)
100g. ⇒ 0 ≤ **card mereo**_E(e) ≤ 1
100h. ∧ **mereo**_N(n) ∈ graph_NIs(g) ∧ **mereo**_E(e) ∈ graph_EIs(g)

This (**obs_**, **uid_** and **mereo_**) observer functions, sets and axiomatic description is sufficient the describe graphs, no need for a Cartesian (of nodes and edges of a grap) – it is, of course, "buried" in the two (nodes and edges) **obs**_erver functions.

9.1.2.3 Routes of Graphs

9.1.2.4 Cyclic and Acyclic Graphs

9.1.2.5 Function Representation of Graphs

One way to represent $\boxed{\text{more to come}}$

9.1.3 Playing Around with Graphs

Example 59 Your Home: We shall outline an informal description of a problem "close to home", one that lends itself to a graph representation.

- Consider a floor plan of Your home, whether it be a free-standing villa in a garden, a farm-building, a row-house with front and back yards, or an apartment, with one or more proper entries (front, 'kitchen" and/or garden/terrace/balcony doors).

- Consider each room in Your home, its possible balconies, the [front and/or back] yard, or, if an apartment, the shared staircase landing – consider them nodes in a graph.
- Consider each door or opening that allows direct access from one room to another, between a room and a balcony, or a yard, or a landing – consider them edges in a graph.

With the above we have presented a case for abstracting what has been described as graphs. We shall not formalize the kind of graphs implied by the above description – but shall pose that as an exercise![1]

We now augment the above description.

- With each room in the house we can associate a number of attributes:
  - Rooms have
    * zero, one or more windows;
    * a certain floor area and floor-to-ceiling height;
    * a certain purpose: entry/exit, corridor, kitchen, toilet, bath, living, dining, sleeping, office, laundry, all-purpose, balcony, terrace, etc.;
    * or may be designated certain family members: parents, children, pets (i.e., animals), or other;
    * et cetera.
  - Doors and openings have
    * one- or two-way access, entry/exit;
    * width and height;
    * blinded or glass;
    * et cetera.

In modeling the above we may consider the model to be a domain model of homes. In so doing we may model rooms, doors and openings as parts, cf. Sect. **1.6.1.3** on page 31, and attributes as *internal qualities*, cf. Sect. **1.6.2.3** on page 39 of Chapter **1**. Rooms, doors and openings, further, have unique identification, cf. Sect. **1.6.2.1** on page 36 and have mereologies, cf. Sect. **1.6.2.2** on page 38 that informs us as to which rooms are connect with which other rooms etc. We pose, as an exercise, the modeling of `Your Home` in the style of *domain model*, using the domain modeling tools of **obs_** [**endurant_**], **uid_**, **mereo_** and **attr_** observers.[2] ■

Example 60 Your Neighbourhood: We shall outline an informal description of a problem "close to home", one that lends itself to a graph representation.

[1] **Editorial note:** – to be inserted!
[2] **Editorial note:** – to be inserted

- Consider Yourself to live at an address, somewhere.
- Let an address indicate either a "point" on a link immediate between two hubs, that is an intersection between two immediately adjacent links, i.e., street segment.
- Consider Your closest family, friends, colleagues, acquaintances, etc., not living at Your address but at other addresses reachable by foot – say within an hours walk – consider them the addresses of contacts.
- Consider also the contact addresses of all the places You otherwise encounter in Your daily life: schools, workplaces, shops, libraries, public offices, etc.
- Now let the addresses of all these people and institutions be nodes of a graph.
- And then, first, let the different addresses
-
-

We shall not formalize the kind of graphs implied by the above description – but shall pose that as an exercise![3] ■

Example 61 Road Transport Graphs: We shall outline an informal description of a problem of everyday experience, one that lends itself to a graph representation. We have chosen to model multiplicities of sub-trees as Cartesians – the "seeming", left-to-right order, as You "read" the domain description, is in deference to *espaliers*, cf. Fig. **9**.6 on page 130 – but the domain model give next applies to almost all trees, whether natural or conceptual.

101. The domain is *road transport*, RT.
102. From a road transport we can observe a Cartesian of (presently just) two entities: a *road net*, RN, and a set of *automobiles*, SA.
103. From a *road net* we can observe a Cartesian of (presently just) two entities: a set of *hubs*, SH, (i.e., street intersections, or important "stops" along a street), and a set *links*, SL, (i.e., street segments between immediately "neighbouring" hubs).
104. We presently consider hubs and links atomic.
105. From hubs, links and automobiles we can observe (**uid_**) that all hubs, links and automobiles are *uniquely identifiable.*

    And from hubs and links we can observe (**mereo_**) their *mereology*:

[3] **Editorial note:** – to be inserted!

106. The mereology of a hub is the set of zero, one or more unique link identifiers. If the set is empty, the hub is "isolated": no streets lead into it or out from it. If the set is singleton: {li}, then a link emanates from and "lops" back at the hub. The link identifiers must [thus] be of links of the road transport.

107. The mereology of a link is a singleton set of one hub identifier or a set of just two such: if one, the link "loops back" on the hub it thus both emanates from and is incident upon. If two, i.e., the only other alternative, then the link connects the two identified hubs. The hub identifiers must [thus] be of hubs of the road transport.

    From links we can observe their length, LEN; and many other attributes which we [presently][4] shall not cover, likewise for hubs.

108. From automobiles we can observe their location on the road net: at a hub, **attr**_at_Hub, or on a link, **attr**_on_Link.

109. The at_Hub attribute indicates the hub.

110. The on_Link attribute indicates the link, the direction (from→to) it travels on the link, and the *fraction*, fraction, along that link, a real number properly between 0 and 1.

type
101. RT
102. RN
102. SA = A-**set**, A
103. RN = SH × SL
103. SH = H-**set**
103. SL = L-**set**
105. HI, LI, AI
106. HMer = LI-**set**
107. LMer = HI-**set**; **axiom** ∀ l:L•1≤**card mereo**_L(l)=2
108. APos = at_Hub | on_Link
109. atHub :: HI
110. onLink :: LI×(from:HI×fraction:**Real**×to:HI); **axiom** ∀ (__,(__,f,__)):OnLink•0<f<1
value
102. **obs**_RN: RT → RN
102. **obs**_SA: RT → SA
103. **uid**_SH: RN → SH
103. **uid**_SL: RN → SL
105. **uid**_H: H → HI, **uid**_L: L → LI, **uid**_A: A → AI,
106. **mereo**_H: H → HMer

[4]**Editorial note:** Chapter **15** will show time-related link, hub and automobile attributes in Example **??** on page ??.

107. **mereo**_L: L → LMer
108. **attr**_APos: A → APos
axiom
106. ∀ rt:RT •
106. [**Wellformedness of Hub Mereologies**]
106. **let** (_,his) = hub_uids(rt) **in** (his ∩ (link_ids(rt)∪{a}utomobile_ids(rt))) = {} **end**
107. [**Wellformedness of Link Mereologies**]
107. ∧ ∀ h:H := h ∈ obs_SH(obs_RN(rt)) ⇒ **mereo**_H(h) ⊆ link_uids(rt)
107. [**Wellformedness of Automobile Attributes**]
109. ∧ ∀ l:L := l ∈ obs_SL(obs_RN(rt)) ⇒ **mereo**_L(l) ⊆ hub_uids(rt)

The above makes use of auxiliary functions – which all yield a pair: a well-formedness truth value and a set of unique identifiers.

111. hub_uids calculates all the *u*nique *id*entifiers of hubs of a road transport.

112. So it retrieves all hubs.

113. And yields the set of all their identifiers, while securing that the automobiles have distinct identifiers.

111. hub_uids: RT → (**Bool**×HI-**set**)
111. hub_uids(rt) ≡
112. **let** hs = obs_SH(obs_RN(rt)) **in**
113. **let** his = { **uid**_H(h) | h:H • h ∈ hs } **in**
113. (**card** hs=**card** his,ids) **end end**

114. link_uids calculates all the *u*nique *id*entifiers of links of a road transport.

115. Similar to Item 112.

116. Similar to Item 113.

114. link_uids: RT → (**Bool**×LI-**set**)
114. link_uids(rt) ≡
115. **let** ls = obs_SL(obs_RN(rt)) **in**
116. **let** lis = { **uid**_L(l) | l:L • l ∈ ls } **in**
116. (**card** ls=**card** lis,ids) **end end**

117. automobile_uids calculates all the *u*nique *id*entifiers of automobiles of a road transport.

118. Similar to Item 112.

119. Similar to Item 113.

117. automobile_uids: RT $\rightarrow$ (**Bool**$\times$AI-**set**)
117. automobile_uids(rt) $\equiv$
118. **let** as = obs_SH(obs_RN(rt)) **in**
119. **let** ais = { **uid**_A(a) | a:A • a $\in$ as } **in**
119. (**card** as=**card** ais,ids) **end end**

120. No hub identifier equals any link or automobile identifier.

121. No link identifier equals any hub or automobile identifier.

122. No automobile identifier equals any hub or link identifier.

axiom
120. (hub_ids(rt) $\cap$ (link_ids(rt) $\cup$ automobile_ids(rt))) = {}
121. (link_ids(rt) $\cap$ (hub_ids(rt) $\cup$ automobile_ids(rt))) = {}
122. (automobile_ids(rt) $\cap$ (hub_ids(rt) $\cup$ link_ids(rt))) = {} ∎

9.2 Trees

Trees abound! Natural ones and conceptual, abstract ones!

9.2.1 Natural Trees

There are many forms of trees: beech, birch, elms, pines and others resemble one-another wrt. roots, trunks and branchings. They "share" forms of leaves – where pines have needles.

9.2.1.1 A Prototype Tree: The Beech

The beech tree is "The Danish National tree"! See Figs. **9**.3– **9**.4 on the facing page

Figure 9.3: Roots and Trunk

Figure 9.4: Trunks, Branchings and Leaves

9.2.1.2 Banyan Trees

Banyan trees illustrate that one cannot count on all trees each having just one root! A main characteristics of a banyan tree is its many roots[5] See Fig. **9**.5.

Figure 9.5: Banyan Trees

9.2.1.3 Espaliers

Espalier is the horticultural and ancient agricultural practice of controlling woody plant growth for the production of fruit, by pruning and tying branches to a frame.[6] See Fig. **9**.6.

[5] A banyan is a fig that develops accessory trunks from adventitious prop roots, allowing the tree to spread outwards indefinitely. This distinguishes banyans from other trees with a strangler habit that begin life as an epiphyte, i.e., a plant that grows on another plant, when its seed germinates in a crack or crevice of a host tree or edifice. The many banyan species include: `Ficus microcarpa`, which is native to *Australia, Bangladesh, Bhutan, China, India, Malay Archipelago, Nepal, New Caledonia, New Guinea, Pakistan, Ryukyu*

Figure 9.6: Espaliers

9.2.2 Conceptual Trees

9.2.2.1 Genealogy

A genealogy (from Ancient Greek $\gamma\epsilon\nu\epsilon\alpha\lambda o\gamma\iota\alpha$ (genealoga) *"the making of a pedigree"*) is the study and pictorialisation of families, family history, and the tracing of their lineages. See Fig. **9**.7.

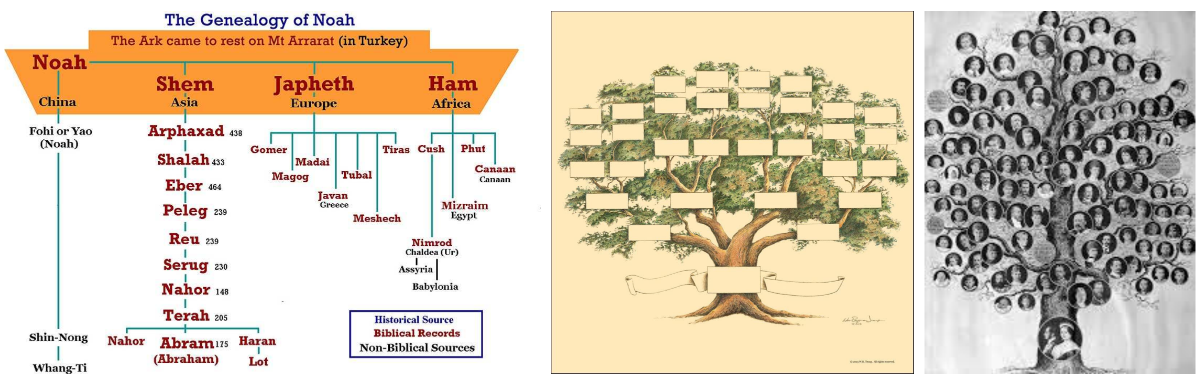


Figure 9.7: Some Genealogies

Islands, *Southeast Asia*, *Sri Lanka* and *Taiwan*, and is a significant invasive species elsewhere `[Wikipedia]`.

[6] Plants are frequently shaped in formal patterns, flat against a structure such as a wall, fence, or trellis, and also plants which have been shaped in this way. A horizontal espalier Free-standing espaliered fruit trees (step-over) at Standen, West Sussex. The trees are used to create a fruit border or low hedge.

Espaliers, trained into flat two-dimensional forms, are used not only for decorative purposes, but also for gardens in which space is limited. In a temperate climate, espaliers may be trained next to a wall that can reflect more sunlight and retain heat overnight or oriented so that they absorb maximum sunlight by training them parallel to the equator. These two strategies allow the season to be extended so that fruit has more time to mature.

A restricted form of training consists of a central stem and a number of paired horizontal branches all trained in the same plane. The most important advantage is that of being able to increase the growth of a branch by training it vertically. Later, one can decrease growth while increasing fruit production by training it horizontally `[Wikipedia]`.

9.2.2.2 Ontologies and Taxonomies

We refer to Sect. **0.8** on page 11. See Fig. **9**.8 for two examples. See also Figs. **1**.1 on page 27 and **??** on page ??[7]

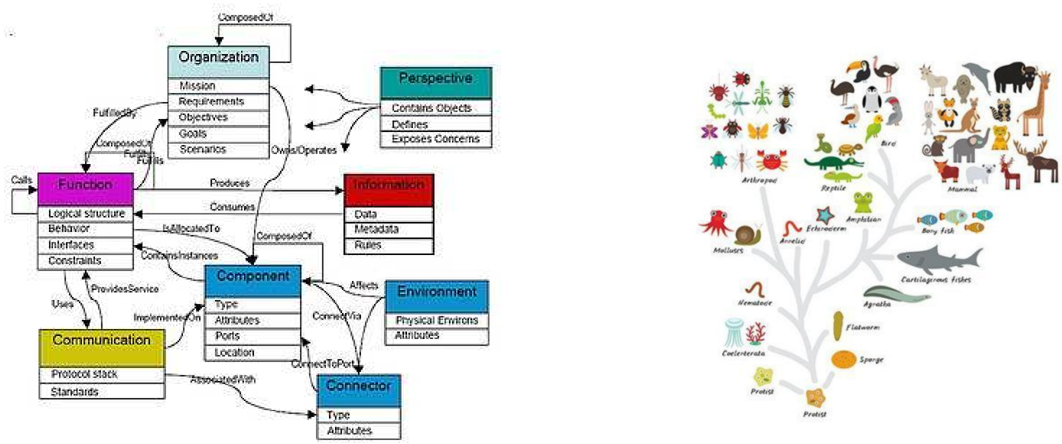


Figure 9.8: An Ontology and a Taxonomy

9.2.3 Cartesian Trees

We shall illustrate a generic model of trees. The model focus on the components and the *syntactic*[8] structure of simple trees.

Example 62 Cartesian Trees:
Narration:

123. Let tree (T) *roots* (R), *branchings* (B) and *leaves* (L), be modeled by some, presently further undescribed types.

124. Let tree *trunks* [the "straight part of a tree between forks], modeled by type Trunk, be some other, presently further undescribed type.

125. Let Ti stand for *branches* into exactly n trunks – where n ranges from 1 to, let, say, 5![9].

---

[7] **Editorial note:** To be inserted!

[8] By *syntax* we shall mean the arrangement of elements (e.g., words or parts) and their composition (e.g., phrases or composite parts) to create well-formed structure (e.g., sentences or parts) in a language or model. [By words and phrases we mean those of a (written/spoken) laguages; and by parts we mean those of a domain model.]

[9] Just a choice, to make things "describable". It is not many tree forks that branch into more than that number!?

126. A *sub-tree* tree, ST, is
    (a) either a *branching* **and** from 1 to n trunks, Ti, i.e., T1,T2, ..., Tn.
    (b) or a set of zero, one or more presently further unspecified *leaves, fruits and seeds* (LFS).
127. *Trees* are now "further described" as the composition of a root, r:R and a sub-tree st:ST.
128. Roots and branchings have *diameter* attributes (D). For roots that diameter is measured at a suitable "uppermost" level.
129. Trunks have *length* (TL), *i*nitial (ID) and *f*inal (FD) *diameters*.
130. From a leaf, fruit and seed we can observe the following attributes: Whether a seasonal or all-year leaf, what kind of fruit (*apple, cherry, orange, peach, pear*, etc., and "within these their sub-types[10].

Formalisation:

type
123. T, R, B, L
124. Trunk
125. Ti = T1 | T2 | ... | Tn
125. T1 :: Trunk
125. T2 = Trunk × Trunk
125. ...
125. Tn = Trunk × Trunk × ... × Trunk, **axiom** n [is typically] ≤ 5
126. ST =
126a. B × Ti
126b. | L-**set**
127. T = R × ST
128. D = **Nat cm**
129. TL = **Nat cm**, ID = **Nat cm**, FD = **Nat cm**
130. LFS = L | F | S
130. FruitType = Apple | Cherry | Orange | Peach | Pear | ...
130. AppleType = Ambrosia | Braeburn | BelleDeBoskoop | ...
130. ...
value
128. **attr**_D: (R|B) → D,
129. **attr**_TL: Trunk → TL, **attr**_ID: Trunk → ID, **attr**_FD: Trunk → FD ■

[10] For apples there are, e.g., at least these 26 sub-types: *Ambrosia, Braeburn, Belle de Boskoop (also called Goudrenet, etc.), Cameo, Empire, Envy, Fuji, Gala, Golden Delicious, Granny Smith, Gravenstein, Hidden rose, Holstein, Honeycrisp, Jazz, Jonagold, Lady Alice, Liberty, McIntosh, Mutsu, Opal, Pacific rose, Pazazz, Pink lady, Red Delicious, Winesap.*

130. **attr**_L: LFS $\rightarrow$ ...
130. **attr**_FruitType: LFS $\xrightarrow{\sim}$ FuitType
130. **attr**_AppleType: LFS $\xrightarrow{\sim}$ AppleType
130. ...

Applying the attribute observers **attr**_FruitType and **attr**_AppleType to a leaf/fruit/seed entitity which is not a fruit, respectively which is not an apple, results in **chaos**, hence the attribute functionality is partial ($\xrightarrow{\sim}$).

That's it! The narrative and the commensurate formalisation says it all! The final arbiter is the formalisation.

In Sect. 66 on page 141 and Example 68 on page 150 we shall discuss alternative tree models – as simple modifications of the present model ■

9.3 Closing

to come

9.3.1 Summary

to come

9.3.2 Conclusion

to come

9.4 Exercises

Exercise 31 Your Home:

Exercise 32 Your Neighbourhood:

Exercise 33 Properties of Trees: *Based on the formalization of Example 62 on page 131 define functions that (1.) calculate the the set of all paths from the root of a tree to all its leaves, (2.) the (set of) shortest and logest path(s), and (3.) the number of leaves!*

Chapter 10

Lists

Contents

In this chapter we shall introduce the programming cum modelling language MoLa's concept of **lists**.

Motivation: Lists

Humans tend to order certain phenomena or even abstract ideas, concepts. This ordering may be total, in which case we call them lists. (If they are partial we may refer to them as *trees* or *lattices*.)

We list the days of a week by <Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday>. We list the months of a year by <January, February, March, April, May, June, July, August, September, October, November, December>. We list the days of a month by <1, 2, ..., 28>, or <1, 2, ..., 29>, <1, 2, ..., 30>, or <1, 2, ..., 231>, according to the month, and we list the years of our AC calendar by <0, 1, 2, ..., 2023> and so forth. We say that Monday is the first day of the week and Sunday is the 7th day of the week; that January is the 1st month of the year, July the 7th month, and so forth; and so forth.

Definition 77 List: *By a list we shall mean the same as by a sequence, or tuple: an ordered, i.e., an indexed (or indexable), grouping of zero, one or more — not necessarily distinct entities — all being of a common type, i.e., of a type that can be named. Furthermore, for the "thing" to be classified as a list it must be meaningful to speak of such list operations as the head,* **hd,** *the tail,* **tl,** *the distinct elements,* **elems,** *the set of all the indices,* **inds,** *the* **len***gth, and of selecting an i'th element of a list $\ell(i)$, of concatenating, $\frown$, two lists, and of inquiring whether two lists are equal (not equal), $=(\neq)$.*

10.1 List Presentations

The present section is structured into two subsections, as are Sects. **3.2** and **11.1**.

10.1.1 List Enumeration

Let a_1, a_2, ..., a_n stand for some distinct elements of type A, then

value ⟨ a_1, a_2, ..., a_n ⟩

expresses an informal, abstract way of *explicitly enumerating* a list, of type A^*, of n elements. It is the use of further unidentified a_is and the ellipsis: "..." that makes the presentation informal and abstract. A formal, concrete way would be:

Example 63 A Row of Fruits: Let a_1, a_2, a_3 stand for three distinct $a$pples, p_1, p_2 for two distinct $p$ears, and o for a single orange, then

value ⟨ a_1, a_2, p_2, p_1, a_3, o ⟩

then exemplifies an arbitrarily, linearly ordered basket, a list, of fruits ∎

10.1.2 List Comprehension

Let A be some type with elements a_1, a_2, ..., a_n, ... and let aset be a finite or infinite set of element in A. Let p:P be a predicate over elements of A, and let f:F be a function over [perhaps not all] a:A and, optionally, natural numbers i:**Nat** yielding elements b of type B. Finally let the [optional] expression m≤i≤n express an ordering of [let us call them] indices i. Then the last line in

type
A, B
value
aset = A-**set**, p: A→**Bool**, f: A×**Nat**$\xrightarrow{\sim}$B, m,n:**Nat**
⟨ f(a)(i) | a:A[,i:**Nat**] • a ∈ aset ∧ p(a) [∧ m≤i≤n] ⟩

expresses a *list comprehension*. In short: it denotes the list of all those f(a) of type B for which the property p(a) holds – and, optionally, listed in the order m≤i≤n.

Precondition: If aset is infinite the optional expression m≤i≤n is omitted, the q function reduces to q: A→B and the ordering of the bs is arbitrary.

The signs ⟨ and ⟩ can be said to form and delineate the list. The | separates the text between the ⟨ and ⟩ into two texts. To the left of | is an expression, here just q(...). To the right of | there are two texts separated by a •. Between | and • the clause defines the type of a, hence its "larger" range, and its actual range aset. Between • and ⟩ the as are limited here to within aset, and the predicate clause, p(a), delimits the as to those which satisfy that predicate, i.e., for which a holds, i.e, is **true**.

Example 64 Simple List Examples: Let fact name the factorial function[1], then

⟨ fact(1),fact(2),fact(3),fact(4),fact(5),fact(6) ⟩

expresses a simple list of six elements, the first six factorials. So does:

⟨ fact(i) | i:**Nat** • 1≤i≤6 ⟩ ▪

10.2 List Types

Let A stand for a type whose possibly infinite number of elements include $\{a_1, a_2, \ldots, a_n, \ldots\}$.

Types whose values can be considered finite, respectively finite or infinite lists of A elements can be defined using the suffix * and $^\omega$ type operators, respectively:

type
A
F = A^*
F = A^ω

The above expressions A^* and A^ω are *list type expressions*

10.3 List Operations

We define the list operations: **hd, tl, elems, inds, len,** list element selection $\ell(i)$, concatenation $\frown$, equality =, and inequality ≠.

[1]fact was defined in Example **??** on page ??.

10.3.1 Operation Signatures

Signatures:

value
  hd: $A^* \to A$
  tl: $A^* \to A^*$
  len: $A^* \to$ **Nat**
  inds: $A^* \to$ **Nat-set**
  elems: $A^* \to$ A-**set**
  .(.): $A^* \times$ **Nat** $\to A$
  $\widehat{\ }$: $A^* \times A^* \to A^*$
  $=$: $A^* \times A^* \to$ **Bool**
  $\neq$: $A^* \times A^* \to$ **Bool**

Examples:

hd$\langle$a1,a2,...,am$\rangle$=a1
tl$\langle$a1,a2,...,am$\rangle$=$\langle$a2,...,am$\rangle$
len$\langle$a1,a2,...,am$\rangle$=m
inds$\langle$a1,a2,...,am$\rangle$=$\{$1,2,...,m$\}$
elems$\langle$a1,a2,...,am$\rangle$=$\{$a1,a2,...,am$\}$
$\langle$a1,a2,...,am$\rangle$(i)=ai
$\langle$a,b,c$\rangle\widehat{\ }\langle$a,b,d$\rangle$ = $\langle$a,b,c,a,b,d$\rangle$
$\langle$a,b,c$\rangle$=$\langle$a,b,c$\rangle$
$\langle$a,b,c$\rangle \neq \langle$a,b,d$\rangle$ ■

10.3.2 Informal Operation Definitions

hd (head) and **tl** (tail) are assumed primitive operations.

value
  len q $\equiv$ **if** q = $\langle\rangle$ **then** 0 **else** 1 + **len tl** q **end**

  inds q $\equiv$ $\{$i|i:**Nat**•1$\leq$i$\leq$**len** q$\}$

  elems q $\equiv$ $\{$q(i)|i:**Nat**•i $\in$ **inds** q $\}$

  q(i) $\equiv$ **if** i=1 **then let** a:A,q′:Q • q=$\langle$a$\rangle\widehat{\ }$q′ **in** a **end else tl** q(i−1) **end**
  pre: i>0 $\wedge$ q$\neq\langle\rangle$

  fq $\widehat{\ }$ iq $\equiv$ $\langle$**if** 1$\leq$ i$\leq$ **len** fq **then** fq(i) **else** iq(i−**len** fq) **end**|i:**Nat**•i$\leq$**len** fq+**len**$\rangle$

  iq′ = iq″ $\equiv$ **inds** iq′=**inds** iq″$\wedge\forall$ i:**Nat**•i $\in$ **inds** iq′$\Rightarrow$iq′(i)=iq″(i)
  iq′ $\neq$ iq″ $\equiv$ $\sim$(iq′ = iq″)

10.4 Playing Around with Lists

10.4.1 Some Preliminary Remarks

Let a, a_i, etcetera be any not necessarily distinct values of some type(s).

  value a:A, ..., a_i:A_i, ...

We assume the definition of ...

10.4.2 Functions and Predicates

10.4.2.1 Modeling Trees

In Sect. **8.5.2.1**, this section, and in Sect. **11.4.2.1** we apply the set, Cartesian, list and map type concepts to the abstract modeling of some form of trees.

We remind the reader of Fig. **8**.1 on page 114.

Example 65 **A List Tree Type:**

131. A list tree, $\mathrm{T}_{\mathcal{C}}$, has a root, a [main] trunk, a branching, ***and***[2] a list of two or more list sub-trees.

132. A root is presently modeled by a [further unspecified] root identifier.

133. A trunk is presently modeled by a name, i.e., a [further unspecified] trunk identifier.

134. A branching is presently modeled by a name, i.e., a [further unspecified] branch identifier.

135. A list sub-tree, $\mathrm{ST}_{\mathcal{C}}$, is ***either*** a leaf ***or***[3] is a *proper* list sub-tree.

136. A leaf is presently modeled by a name, i.e., a [further unspecified] leaf identifier.

137. A proper list sub-tree, $\mathrm{PT}_{\mathcal{C}}$, has a trunk, a branching ***and*** a list of two or more proper list sub-trees.

138. Root, branch, trunk and leaf identifiers are all [further undefined] quantities of the same “kind”, i.e., sort[4].

type
131. $\mathsf{T}_{\mathcal{L}} = \mathsf{RT} \times \mathsf{TR} \times \mathsf{BR} \times \mathsf{ST}_{\mathcal{L}}{}^*$
132. $\mathsf{RT} = \mathsf{RID}$
133. $\mathsf{TR} = \mathsf{TID}$
134. $\mathsf{BR} = \mathsf{BID}$
135. $\mathsf{ST}_{\mathcal{L}} = \mathsf{LF} \mid \mathsf{PT}_{\mathcal{L}}$
136. $\mathsf{LF} = \mathsf{LID}$
137. $\mathsf{PT}_{\mathcal{L}} = \mathsf{TR} \times \mathsf{BR} \times \mathsf{PT}_{\mathcal{L}}{}^*$
137. $\mathsf{ID} = \mathsf{RID} \mid \mathsf{RID} \mid \mathsf{BID} \mid \mathsf{LID}$
axiom
131. $\forall\ (\_,\_,\_,\mathsf{stl}){:}\mathsf{T}_{\mathcal{L}} \bullet\ \textbf{len}\ \mathsf{stl} \geq 2$
137. $\forall\ (\_,\_,\mathsf{ptl}){:}\mathsf{PT}_{\mathcal{L}} \bullet\ \textbf{len}\ \mathsf{ptl} \geq 2$ ■

[2]The textual enumeration, separated by commas and ending with ***and*** informs us that a Cartesian, ×, is being defined.

[3]The textual ***either*** ***or*** informs us that a union type, |, is being defined.

[4]‘sort’ is just another name for ‘type’.

to be written

139.

140.

139.
140.

141.

142.

141.
142.

10.4.2.2 Functions over Lists

Figure **10**.1 informally illustrates a *breadth-first, right-to-left* **traversal** of the tree of Fig. **8**.1 on page 114.

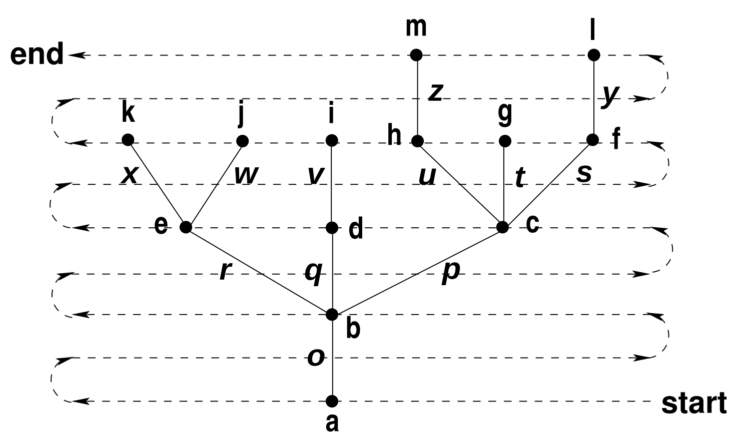


Figure 10.1: A Breadth-first Right-to-Left Tree Traversal

143.

144.

143.
144.

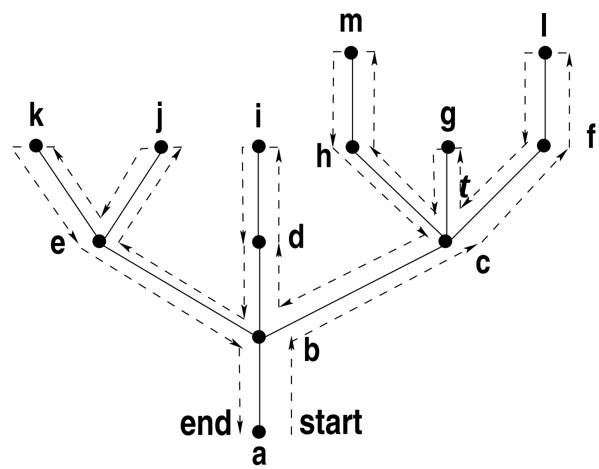


Figure 10.2: An All-order Tree Traversal: ⟨**a,b,c,f,l,f,c,g,c,h,m,h,c,b,d,i,d,b,e,j,e,k,e,b,a**⟩

Figure **10**.2 informally illustrates a right-to-left all-order [depth-first] **traversal** of the tree of Fig. **8**.1 on page 114 in which all nodes are "visited" every time they are 'passed-by', starting from the root.

These are possible "visiting" [depth-first] traversal orders[5]:

- A **pre-order** traversal "visits" a node only on first encounter.
  Example: ⟨**a,b,c,f,l,g,h,m,d,i,e,j,k**⟩
- A **post-order** traversal "visits" a node only on last encounter.
  Example: ⟨**l,f,c,g,m,h,i,d,j,k,e,b,a**⟩
- An **proper in-order** traversal "visits" a branch and root nodes on every encounter after the first and before the last and leaf nodes "whenever".
  Example: ⟨**i,c,,g,c,m,c,b,i,b,j,e,k**⟩
- An **all-order** traversal "visits" a node on all encounters.
  Example: ⟨**a,b,c,f,l,f,c,g,c,h,m,h,c,b,d,i,d,b,e,j,e,k,e,b,a**⟩

145.

146.

145.
146.

Example 66 **Ordered Trees:**

to be written

■

[5]**Editorial note:** Check the examples

10.5 Syntax

By *syntax* we shall mean the arrangement of elements (e.g., words or parts) and their composition (e.g., phrases or composite parts) to create well-formed structure (e.g., sentences or parts) in a language or model. [By words and phrases we mean those of a (written/spoken) laguages; and by parts we mean those of a domain model.]

BNF Grammar: List Expressions

[1.]	<List-Expr>	::=	$\langle\rangle$
[2.]		$\mid$	$\langle$ <Expr-sequence> $\rangle$
[3.]		$\mid$	$\langle$ <Id> $\mid$ <IdTypList> $\bullet$ <Pred-expr> $\rangle$
[4.]		$\mid$	<Pref-List-op> <List-Expr>
[5.]		$\mid$	<List-Expr> <Inf-List-op> <List-Expr>
[6.]	<IdTypList>	::=	<Id>:<Type-expr>
[7.]		$\mid$	<IdTypList>, <Id>:<Type-expr>
[8.]	<Pre-List-op>	::=	**hd** $\mid$ **len** $\mid$ **inds**
[9.]	<Inf-List-op>	::=	$\frown$
[10.]	<Expr-sequence>	::=	<Constant>
[11.]		$\mid$	<Variable>
[12.]		$\mid$	<Expression> , <Expr-sequence>
[13.]	<Type-expr>	::=	...
[14.]	<Pred-expr>	::=	...
[15.]	<Constant>	::=	...
[16.]	<Variable>	::=	<Id>
[17.]	<List-Index-Expr>	::=	<Map-Expr> [<Id>]
[18.]	<Boolean-expr>	::=	<Set-Expr> <Set-relation> <Set-Expr> $\mid$ ...
[19.]	<Set-relation>	::=	$= \mid \neq \mid \subset \mid \subseteq$
[20.]	<Arith-expr>	::=	**card** <Set-Expr> $\mid$ ...

We comment on this grammar:

[1.] The empty list is a list expression.

[2.] A non-empty enumeration of one or more expressions is a list expression.

[3.] A list comprehension is a list expression.

[4.] A pair of a [prefix] list operator and a list expression is an expression.

[5.] A triplet of two list expressions infixed by an infix list operator is a list expression.

As used in [3.], an identifier-type list is

[6.] either a pair of a value name and a type identifier,

[7.] or is a part of such and an identifier-type list.

[8.]

[9.]

[10.]

[11.]

[12.]

[13.]

[14.]

[15.]

[16.]

[17.]

[18.]

[19.]

[20.]

10.6 Closing

to be written

10.6.1 Summary

to be written

10.6.2 Conclusion

to be written

10.7 Exercises

Exercise 34 XLists:

Exercise 35 YLists:

Exercise 36 ZLists:

Chapter 11

Maps

Contents

In this chapter we shall introduce the programming cum modelling language MoLa's concept of **maps**.

Motivation: Maps

Maps are here understood as finite definition set, discrete functions, that is as special sets of pairs of *definition set* elements, i.e., values, and *range set* values – where no two such pairs have the same definition set element.

Such sets of "pairings" occur, perhaps not immediately, in domains, mostly as the result of the observers', Yours and my, abstraction of certain phenomena. Examples are: (i) *in many countries, every person has a unique national identification number*, (ii) *in personnel administrations every employee is uniquely associated with various administrative information: birth-date, address, staff rank, etc.*, and (iii) *in many countries, distinct automobiles are endowed with a number of characteristics: engine [motor] number, owner, and, possibly, insurance registration.*

From Sect **8.1** on page 110 we adapt the driver's license and the passport example as follows:

My Danish Drivers$L$icense maps the following fields: Last$N$ame, First$N$ame, Birth$D$ate, Issue$D$ate, Expiry$D$ate, Issuing$A$uthority, NatlIdNumber, Signature and Photo into respective information.

We can present this as a map:

[LN↦Bjørner,FN↦Dines,BD↦4.10.1937,ID↦...,ED↦...,IA↦StatePolice,NId↦...,S↦...,P↦...]

11.1 Map Presentation

The present section is structured into two subsections, as are Sects. **3.2** and **10.1**.

11.1.1 Map Enumeration

Let a_1, a_2, ..., a_n stand for some distinct elements of type A, and b_1, b_2, ..., b_n for some not necessarily distinct elements of type B, then

value [a_1↦b_1, a_2↦b_2, ..., a_n↦b_n]

expresses an informal, abstract way of *explicitly enumerating* a map of n *definition set* elements and **card**{b_1, b_2, ..., b_n} *range* elements. It is the use of further unidentified a_is and b_js and the ellipsis: "..." that makes the presentation informal and abstract. A formal, concrete way would be:

value [fact(1)↦fib(1),fact(2)↦fib(2),...,fact(6)↦fib(6)]

which expresses a map from the first 6 factorial numbers to the first 6 Fibonacci numbers.

11.1.2 Definition and Range Sets of Maps and Functions

Maps accept argument of their *definition set* and yield results in their *range set*.

By the *definition set* of a function (or a map) is meant the set or argument values for which the function (or map) is well-defined ■

By the *range set* of a function (or a map) is meant the set or result values yielded by the function (or map) when it is applied to argument values of its definition set ■

11.1.3 Map Comprehension

Let A be some type with elements a_1, a_2, ..., a_n, ... and let aset, bset be finite or infinite sets of element in A and B respectively. Let p:P and q:Q be predicates over elements of A, respectively B, and let f:F and g:G be functions over [perhaps not all] a:A, respectively B yielding elements c of type C, respectively D. Then the last line in

type
    A, B
value
    aset:A-**set**, bset:B-**set**, p:A→**Bool**, q:A×B→**Bool**, f:A×B $\stackrel{\sim}{\to}$ C, g:A×B $\stackrel{\sim}{\to}$ D
    [f(a,b)↦g(a,b) | a:A,b:B • a ∈ aset ∧ b ∈ bset ∧ p(a) ∧ q(a,b)]

expresses a *map comprehension*. In short: it denotes the (finite or infinite) map of all those f(a) of type C for which the property p(a) holds mapping into those g(a,b) of type D for which the property q(a,b), i.e., for which q(a,b) holds, i.e, is **true**

The signs [and] can be said to form and delineate the map. The | separates the text between the [and] into two texts. To the left of | is an expression, here just f(a)↦g(b). To the right of | there are two texts separated by a •. Between | and • the clause defines the type of as and bs, its "larger" range. Between • and] the as and bs are limited here to within aset respectively bset, and the predicate clauses, p(a), q(a,b), delimits the as and bs, to those which satisfy that predicate, i.e., for which a, respectively b holds, i.e, is **true**, and hence, for which f(a,b), g(a,b) can be applied.

11.2 Map Type

Let A and B stand for types whose possibly infinite number of elements include $\{a_1, a_2, \ldots, a_n, \ldots\}$ and $\{b_1, b_2, \ldots, b_m, \ldots\}$.

Types whose values can be considered finite maps from A elements into B elements can be defined using the infix $\underset{m}{\to}$ type forming operator:

type
  A, B
  M = A $\underset{m}{\to}$ B

11.3 Map Operations

There are eleven map value related operations: •(•), **dom, rng**, †, ∪, \, /, =, ≠, ≡ and °.

value
$\cdot(\cdot)$: M→A $\stackrel{\sim}{\to}$ B
        example: m(ai)=bi

dom: M→A-**set** [domain of map]
  example: dom [a1↦b1,a2↦b2,...,an↦bn]={a1,a2,...,an}
rng: M→B-**set** [range of map]
  example: rng [a1↦b1,a2↦b2,...,an↦bn]={b1,b2,...,bn}
†: M×M→M [override extension]
  example: [a↦b,a′↦b′,a″↦b″]†[a′↦b″,a″↦b′]=[a↦b,a′↦b″,a″↦b′]
∪: M×M→M [merge ∪]
  example: [a↦b,a′↦b′,a″↦b″]∪[a‴↦b‴]=[a↦b,a′↦b′,a″↦b″,a‴↦b‴]
\: M×A-**infset**→M [restriction by]
  example: [a↦b,a′↦b′,a″↦b″]\{a}=[a′↦b′,a″↦b″]
/: M×A-**infset**− M [restriction to]
  example: [a↦b,a′↦b′,a″↦b″]/{a′,a″}=[a′↦b′,a″↦b″]
=,≠: M×M→**Bool**
  example: m−m, m≠m′
°: $(A \underset{m}{\rightarrow} B)\times(B \underset{m}{\rightarrow} C)\stackrel{\sim}{\rightarrow}(A \underset{m}{\rightarrow} C)$ [composition]
  example: [a↦b,a′↦b′]°[b↦c,b′↦c′,b″↦c″]=[a↦c,a′↦c′]

The **dom** operator, for "domain", stand for "extracting" the ***definition set***, see Sect. **6.2** on page 104, of a map. The **rng** operator, for "range", stand for "extracting" the ***range set***, see Sect. **6.2** on page 104, of a map.

11.4 Playing Around with Maps

11.4.1 Some Preliminary Remarks

Let a, ..., a_i, ..., b, ..., b_j, ... etcetera be any not necessarily distinct values of some type , respectively B.

value
  a:A, ..., a_i:A_i, ...
  b:B, ..., b_i:B_j, ...

We assume the definition of ...

11.4.2 Functions and Predicates

11.4.2.1 Modeling Trees

In Sects. **8.5.2.1**, **10.4.2.1**, and in this section we apply the set, Cartesian, list and map type concepts to the abstract modeling of some form of trees.

We remind the reader of Fig. **8**.1 on page 114.

Example 67 **A Map Tree Type:**

147. A map tree, $\mathrm{T}_{\mathcal{M}}$, has a root, a [main] trunk, a branching, ***and***[1] a map from two or more map sub-tree identifiers to map sub-trees.

148. A root is presently modeled by a [further unspecified] root identifier.

149. A trunk is presently modeled by a name, i.e., a [further unspecified] trunk identifier.

150. A branching is presently modeled by a name, i.e., a [further unspecified] branch identifier.

151. A map sub-tree, $\mathrm{ST}_{\mathcal{M}}$, is ***either*** a leaf ***or***[2] is a ***proper*** map sub-tree.

152. A leaf is presently modeled by a name, i.e., a [further unspecified] leaf identifier.

153. A proper map sub-tree, $\mathrm{PT}_{\mathcal{M}}$, has a trunk, a branching ***and*** a a map from two or more map sub-tree identifiers to map sub-trees.

154. Root, branch, trunk, sub-tree and leaf identifiers are all [further undefined] quantities of the same "kind", i.e., sort[3].

type
147. $\mathsf{T}_{\mathcal{M}} = \mathsf{RT} \times \mathsf{TR} \times \mathsf{BR} \times (\mathsf{TID} \xrightarrow[m]{} \mathsf{ST}_{\mathcal{M}})$
148. $\mathsf{RT} = \mathsf{RID}$
149. $\mathsf{TR} = \mathsf{TID}$
150. $\mathsf{BR} = \mathsf{BID}$
150. TID
151. $\mathsf{ST}_{\mathcal{M}} = \mathsf{LF} \mid \mathsf{PT}_{\mathcal{M}}$
152. $\mathsf{LF} = \mathsf{LID}$
153. $\mathsf{PT}_{\mathcal{M}} = \mathsf{TR} \times \mathsf{BR} \times (\mathsf{TID} \xrightarrow[m]{} \mathsf{PT}_{\mathcal{M}})$
153. $\mathsf{ID} = \mathsf{RID} \mid \mathsf{RID} \mid \mathsf{BID} \mid \mathsf{TID} \mid \mathsf{LID}$
axiom
147. $\forall\ (\_,\_,\_,\mathsf{stm}){:}\mathsf{T}_{\mathcal{M}} \bullet\ \textbf{card dom}\ \mathsf{stm} \geq 2$
153. $\forall\ (\_,\_,\mathsf{ptm}){:}\mathsf{PT}_{\mathcal{M}} \bullet\ \textbf{card dom}\ \mathsf{ptm} \geq 2$ ■

155.

156.

155.
156.

[1] The textual enumeration, separated by commas and ending with ***and*** informs us that a Cartesian, ×, is being defined.

[2] The textual ***either*** ***or*** informs us that a union type, |, is being defined.

[3] 'sort' is just another name for 'type'.

157.

158.

157.
158.

11.4.2.2 Functions over Maps

159.

160.

159.
160.

161.

162.

161.
162.

Example 68 **Indexed Trees:**

$\boxed{\text{to be written}}$

■

11.5 Syntax

[4]

BNF Grammar: Map Expressions

<table>
<tr><td>[1.]</td><td><Map-Expr></td><td>::=</td><td>[ ]</td></tr>
<tr><td>[2.]</td><td></td><td>|</td><td>[<Id$_d$ > $\mapsto$ <Id$_r$ >]</td></tr>
<tr><td>[3.]</td><td></td><td>|</td><td>[<Id> | <IdTypList> • <Map-expr>]</td></tr>
<tr><td>[4.]</td><td></td><td>|</td><td><Pref-List-op> <Map-Expr></td></tr>
<tr><td>[5.]</td><td></td><td>|</td><td><List-Expr> <Inf-List-op> <List-Expr></td></tr>
<tr><td>[6.]</td><td><IdTypList></td><td>::=</td><td><Id>:<Type-expr></td></tr>
<tr><td>[7.]</td><td></td><td>|</td><td><IdTypList>, <Id>:<Type-expr></td></tr>
<tr><td>[8.]</td><td><Pre-Map-op></td><td>::=</td><td>hd | len | inds</td></tr>
<tr><td>[9.]</td><td><Inf-Map-op></td><td>::=</td><td>$\widehat{\ }$</td></tr>
<tr><td>[10.]</td><td><Expr-sequence></td><td>::=</td><td><Constant></td></tr>
<tr><td>[11.]</td><td></td><td>|</td><td><Variable></td></tr>
<tr><td>[12.]</td><td></td><td>|</td><td><Expression> , <Expr-sequence></td></tr>
<tr><td>[13.]</td><td><Type-expr></td><td>::=</td><td>...</td></tr>
<tr><td>[14.]</td><td><Pred-expr></td><td>::=</td><td>...</td></tr>
<tr><td>[15.]</td><td><Constant></td><td>::=</td><td>...</td></tr>
<tr><td>[16.]</td><td><Variable></td><td>::=</td><td><Id></td></tr>
<tr><td>[17.]</td><td><Map-Index-Expr></td><td>::=</td><td><Map-Expr> [<Nat-Number-Expr>]</td></tr>
<tr><td>[18.]</td><td><Boolean-expr></td><td>::=</td><td><Set-Expr> <Set-relation> <Set-Expr> | ...</td></tr>
<tr><td>[19.]</td><td><Set-relation></td><td>::=</td><td>$=|\neq|\subset|\subseteq$</td></tr>
<tr><td>[20.]</td><td><Arith-expr></td><td>::=</td><td>card <Set-Expr> | ...</td></tr>
</table>

11.6 Closing

to be written

11.6.1 Summary

to be written

11.6.2 Conclusion

to be written

[4] **Editorial note:** The BNF Syntax for Map Expressions has been copied from Sect. **10.5** on page 142. It is to be final-edited!

11.7 Exercises

Exercise 37 **XMaps:**

Exercise 38 **YMaps:**

Exercise 39 **ZMaps:**

Chapter 12

Types and Sorts

Contents

In this chapter we shall complete our basic treatment of the MoLa programming cum modelling language's concepts of **types** and **sorts**.

Motivation: Types and Sorts

Mankind appears to have been expressing its curiosity with respect to the phenomena that surrounds us in many ways. One such way has been to *classify* phenomena observed: *plants, animals, rivers, mountains,* etcetera.

Here we observe *a "body" of water*, and that phenomenon is then further classified into either being an *ocean*, an *inland sea*, a *lake*, a *river*, a *spring*, a *canal*, or a *swimming pool*!

And so forth.

In Sect. **12.2** we shall establish an **ontology**, see Fig. **12**.2 on page 160, for guiding us in analyzing the physical, manifest world around us – that world of which it is meaningful to say that main components are [more-or-less] man-made, that is, artefactual.

The Swedish botanist, zoologist, taxonomist, and physician, Carl Linnaeus (1707–1778), formalized binomial nomenclature, the modern system of naming living species, especially plants. He is known as the "father of modern taxonomy" [92].

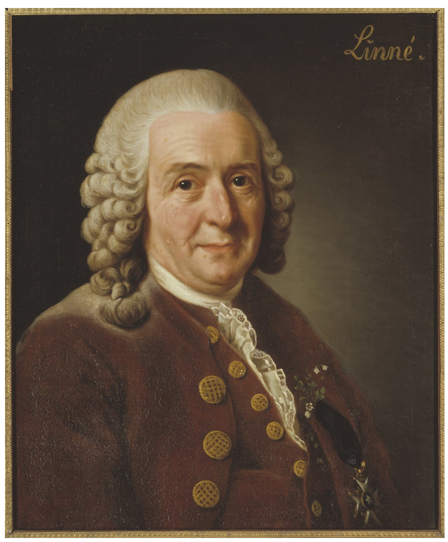


Figure 12.1: Carl Linneaus

Definition 78 Type: *By a* type *we shall initially understand a special class*[1] *of values* ■

Definition 79 Value: *By a* value *we shall initially understand a "thing" – whether a mathematical object or a real world phenomena – something that can be ascribed a* name *and can be claimed to belong to, to be of some,* type ■

[1] We say 'initially', as we shall later indicate that types – as used in this book – are rather special kinds of sets.

12.1 Types

12.1.1 Enumeration of Simple Types

So far we have introduced some simple MoLa types:

- Booleans: Chap. **2**,
- Sets: Chap. **3**,
- Numbers: Chap. **4**,
- Characters &c: Chap. **5**,
- Functions: Chap. **6**,
- Cartesians: Chap. **8**,
- Lists: Chap. **10**, and
- Maps: Chap. **11**.

The function type ($A \to B$ and $A \stackrel{\sim}{\to} B$) was introduced in Chapter **6** – but will be further elaborated upon in this and later chapters.

12.1.2 Type Expressions

Definition 80 **Type Expression:** *By a* type expression *we shall understand a syntactic construct which denotes a type. The syntax of a type expression spans from just being a type literal (***Char, Text, Bool, Num, Nat, Int, Real** *or* **Unit**[2]*), or an identifier, hence a type name, to an expression involving type forming operators* ■

Definition 81 **Type-forming Operator:** *By a* type-forming operator *we shall understand either of the following pre-, inf-, suffix, and distributed-fix operators:* **-set**, **-infset**, $\times$, *, $^\omega$, $\overrightarrow{m}$, $\to$, $\stackrel{\sim}{\to}$, $|$, *and* $\{|...|\}$ ■

We elaborate on this:

- **Char, Text, Bool, Nat, Int, Real** and **Unit** are (atomic) type expressions,

  They denote *characters* (**'a'** etc.), *texts* (**"abc"** etc.), the *Booleans* (**true, false**), the natural numbers (0, 1, 2, ...), the integers (..., -2, -1, 0, 1, 2, ...), the reals (rational and irrational numbers) and ().

- Arbitrarily, i.e., user-chosen, identifiers, like T, stand for type identifiers, if they have been introduced as atomic expressions in the right hand side of a type definition.

So these are the type expressions, TE, of MoLa:

Char,	TE-**set**,	**Int**,
Text,	TE-**infset**,	**Real**,
Bool,	**Nat**,	TId,

[2]The type **Unit** is first introduced in Chapter **17**

TE×...×TE,	TE $\overrightarrow{m}$ TE,	TE|...|TE,
TE*,	TE→TE,	{|...|}, and
TE$^\omega$,	TE$\xrightarrow{\sim}$TE,	**Unit**,

where TId is a type name, see Sect. **12.1.3.1**, and {|...|} is a sub-type expression, see Sect. **12.1.3.2** on the next page, are all type expressions.

12.1.3 Type Definitions

Definition 82 Type Definition: *By a* type definition *we shall understand either the* type introduction *of a type by just naming it or the full* type description *of the type by both naming it and ascribing it some properties:*

type *T* *Type introduction*

type *T* = *Type-Expression* ■ *Type description*

Both of the above forms are MoLa *specification units.*

12.1.3.1 Type Names

We name types[3] for reason of referencability. By a

Definition 83 Type Name: *By a* type name *we shall understand a literal, or an* identifier*. We usually, by convention, start type names with an upper case letter. Type names denote types. When these types are of syntactic quantities, such as elements of a [programming or command] language, then we, again by convention, suggest that the subsequent alphabetic characters of the type identifier are in lower case. For what we might consider semantic types we suggest all uppercase alphabetic characters. Sometimes we indulge in putting in "underlines": '_', properly between characters, and sometimes we also "tail" the identifier with a digit (or two!)* ■

These are the type literals introduced so far, or to be introduced (in Chapter **17**).

- **Char**,
- **Text**,
- **Bool**,
- **Nat**,
- **Int**,
- **Real**, and
- **Unit**.

The type literal **Unit** will be introduced in Chapter **17**.

[3]The concept of types was most recently covered in Sect. **5.1.3.3** on page 92

12.1.3.2 Subtypes

The type forming distributed-fix operator, {|...|}, allows us to define subtypes.

Definition 84 **Subtype:** *A type T_1 is a* subtype *of another type T_2 if all the values contained in T_1 are also contained in T_2. The type T_2 may also contain values that are not in T_1* [47, *Chapter 11, Page 83*] ■

The form of subtype expressions is:

{| binding : type_expr • value_expr |}

Here binding is an *identifier pattern* , i.e., a simple form of just a value identifier, or a parenthesized sequence of [value] identifiers. For example:

id and (id_1,id_2,...,id_n).

type_expr, correspondingly, is a single type identifier, or a Cartesian sequence of type identifiers. For example:

TId, or $\mathsf{TId}_1 \times \mathsf{TId}_2 \times \cdots \times \mathsf{TId}_n$.

And, finally, Boolean_expr is a Boolean expression whose value is "bound" to either id, or, if a Cartesian, to respective id_is in (id_1,id_2,...,id_n).

Example 69 **Subtype of Factorials:**

type Fac = {| f : **Nat** • ∃ n:**Nat** • f=fact(n) }

Where we assume that the fact function is defined elsewhere ■

Example 70 **Subtype of Prime Numbers:**

type Prim = {| p : **Nat** • is_prime(p) }

Where we assume that the is_prime predicate is defined elsewhere ■

12.1.4 Examples

to be written

12.1.5 Recursive Types

So far we have not considered whether, in *type descriptions*, the left-hand side type identifier may occur in the right-hand side type expression. We refer to such type descriptions as *recursive type descriptions*, and to their meaning as *recursive types.*

12.1.5.1 Some Experiments

Let us "experimentally" consider some examples of recursive type descriptions:

type
[1.] AS = AS-**set**
[2.] AC = AC×AC
[3.] AL = AL*
[4.] AM = AM $\overrightarrow{m}$ AM

What do we [intend to] mean by these?

[1.] Any AS-**set** defines {} as one element in AS-**set**.
So if {} is one such element, then it should follow that {{}} is another such element;
that that {{{}}}, ..., {{{{...}}}}, are more such;
and that $\{\{\},\{\{\}\},...\}$ are further such; etc.!
So there is a "solution to **type** AS = AS-**set**.
But is it meaningful?

But:

[2.] For **type** AC = AC×AC is that meaningful?
There is no *"empty"* Cartesian element.
{}, ⟨⟩ and [ ] are "vacuous" i.e., empty] set, list and map values!
So we "rule out" the use of recursion over Cartesians!

However:

[3.-4.] We repeat the [1.] arguments for [3.] and [4.], recursive list and map descriptions.
So we rule them "in"!

But, really, who cares about this "equilibrism". It is a play with mathematics![4] The meaning of the above simplest form of recursive descriptions seems to not match phenomena in a realistic world.

But should we give up recursion!

What about the following type descriptions – where type B is defined elsewhere:

type
[5.] AS = B | AS-**set**
[6.] AC = B | AC×AC
[7.] AL = B | AL*
[8.] AM = B | AM $\overrightarrow{m}$ AM

Now in addition to the "vacuous" solutions there are seemingly meaningful ones also.

[4] Yes, mathematics do provide a solution to the set, list and map recursions, even though the solution implies infinite recursion, i.e., infinitely ongoing "embeddings"!

$\boxed{\text{to come}}$

Example 71 Recursive Phenomena:

Trees:

163. A set-modeled tree is either "gone out", i.e., no longer a tree, or it is a set of set-modeled tree trunks, each being a set-modeled tree or a leaf!

164. A Cartesian-modeled tree is a Cartesian of exactly two Cartesian-modeled tree trunks, each being a Cartesian-modeled tree or a leaf!

165. A list-modeled tree is either "gone out", i.e., no longer a tree, or it is a list of list-modeled tree trunks, each being a list-modeled tree or a leaf!

166. A map-modeled tree is either "gone out", i.e., no longer a tree, or it is a list of map-modeled tree trunks, each being a map-modeled tree or a leaf!

Phone Directories:

167. A phone directory, PD, say on Your mobile phone, uniquely maps the unique combination of phone owners identity, UOI, to the numbers, PN, of their phones, one or more!

168. We leave out describing phone owners identity and phone numbers.

File Directories:

169. A file directory maps names of files into files.

170. A file may be either a simple, say text files, or a file directory.

type

Trees:

163. TS = (TST|L)-**set**, TST = TS | L
164. TC = (TCT×TCT), TCT = TC | L
165. TL = TLT*, TLT = TL | L
166. TM = TMT*, TMT = TM | L

Phone Directories:

167. PD = UOI $\overrightarrow{m}$ PN-**set**
168. UOI, PN

File Directories:

169. FD = FI $\overrightarrow{m}$ FILE
170. FILE = TextFiles | FD
170. TextFiles

12.2 Sorts: Domain Types

There is an altogether different approach to types.

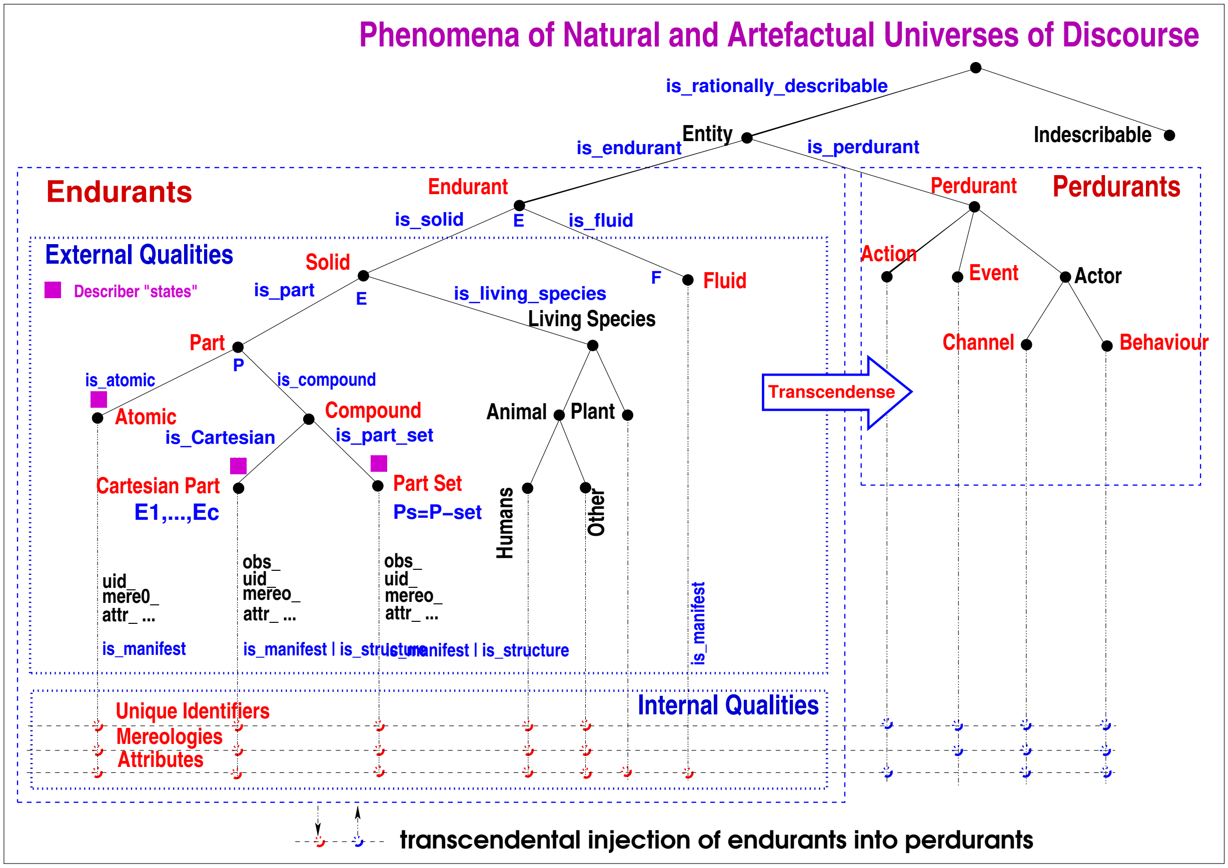


Figure 12.2: A Domain Analysis Ontology

12.2.1 Sorts

12.2.2 Observable Sorts

12.2.2.1 External Qualities

12.2.2.2 Internal Qualities

12.2.2.2.1 Unique Identification

12.2.2.2.2 Mereology

12.2.2.2.3 Attributes

12.3 Closing

to be written

12.3.1 Summary

$\boxed{\text{to be written}}$

12.3.2 Conclusion

$\boxed{\text{to be written}}$

12.4 Exercises

Exercise 40 **XTypes:**

Exercise 41 **YTypes:**

Exercise 42 **ZTypes:**

Part III

Space and Time

Contents

Chapter 13

Space

Contents

13.1 Space Motivated Philosophically

We motivate the concept of indefinite space as follows.

[86, pp 154] *"The two relations* *asymmetric* *and* *symmetric*, *by a transcendental deduction, can be given an interpretation:* The relation (spatial) *direction* is asymmetric; and the relation (spatial) *distance* is symmetric. Direction and distance can be understood as spatial relations. From these relations are derived the relation *in-between*. Hence we must conclude that we exist in space.

Space is therefore an unavoidable characteristic of any possible world".

From the direction and distance relations one can derive *Euclidean Geometry*.

There is but just one space. It is all around us, from the inner earth to the farthest galaxy. It is not manifest. We can not observe it as we observe a road or a human.

13.2 The $\mathbb{SPACE}$ Type

171. There is an abstract notion of (definite) $\mathbb{SPACE}$(s) of further unanalysable points;

172. and there is a notion of $\mathbb{POINT}$ in $\mathbb{SPACE}$.

type
171 $\mathbb{SPACE}$
172 $\mathbb{POINT}$

Space is not an attribute of endurants. Space is just there. So we do not define an observer, `observe_space`. For us, bound to model mostly artifactual worlds on this earth there is but one space. Although $\mathbb{SPACE}$, as a type, could be thought of as defining more than one space we shall consider these isomorphic!

13.3 Spatial Observer

173. A point observer, observe_$\mathbb{POINT}$, is a function which applies to physical endurants, e, and yield a point, $\ell : \mathbb{POINT}$.

value
173 observe_$\mathbb{POINT}$: E $\rightarrow$ $\mathbb{POINT}$

Where the observe_$\mathbb{POINT}$(e) is "taken" we leave up to You! $\mathbb{POINT}$ could be measured, here "on earth", in terms of a triplet: (latitude,longitude,altitude)[1]:

- latitude: is a distance east or west of the prime meridian[2].
- longitude: a coordinate that specifies the northsouth position of a point on the surface of the Earth or another celestial body[3].
- altitude: is a distance measurement, usually in the vertical or "up" direction, between a reference datum and a point or object[4].

[1] `https://en.wikipedia.org/wiki/Geographic_coordinate_system` and `https://en.wikipedia.-org/wiki/Spatial_reference_system`

[2] Lines of longitude, also called meridians, are imaginary lines that divide the Earth. They run north to south from pole to pole, but they measure the distance east or west. Longitude is measured in degrees, minutes, and seconds [`https://oceanservice.noaa.gov/facts/longitude.html`].

[3] Latitude is given as an angle that ranges from 90 at the south pole to 90 at the north pole, with 0 at the Equator. Lines of constant latitude, or parallels, run eastwest as circles parallel to the equator. Latitude and longitude are used together as a coordinate pair to specify a location on the surface of the Earth [`https://en.wikipedia.org/wiki/Latitude`].

[4] The exact definition and reference datum varies according to the context (e.g., aviation, geometry, geographical survey, sport, or atmospheric pressure). Although the term altitude is commonly used to mean the height above sea level of a location, in geography the term elevation is often preferred for this usage [`https://en.wikipedia.org/wiki/Altitude`].

13.4 Models of Technical Drawings

Technical drawings[5], from public authorities cadestral maps, via road net authorities cartographic, i.e., accurate maps of any road segment, car manufacturers drawings of automobile parts, to architects building plans, with their annotations, are intended to be *"to scale"*, i.e., to be accurate enough the be precise descriptions.

Technical drawings, are models themselves, so, in that sense, there is no reason to model a model!

For that reason we suggest to not model technical drawings otherwise!

Financial service institutions, i.e., banks, insurance companies, stock trading, transportation, whether of automobiles, ships, trains or aircraft, etc., have no such models – so we must study models of them in order to understand their domains.

13.5 Spatial Concepts

We suggest, besides $\mathbb{POINT}$s, the following spatial attribute possibilities:

174. $\mathbb{EXTENT}$ as a dense set of $\mathbb{POINT}$s;
175. Volume, of concrete type, for example, m^3, as the "volume" of an $\mathbb{EXTENT}$ such that
176. $\mathbb{SURFACE}$s as dense sets of $\mathbb{POINT}$s have no volume, but an
177. Area, of concrete type, for example, m^2, as the "area" of a dense set of $\mathbb{POINT}$s;
178. $\mathbb{LINE}$ as dense set of $\mathbb{POINT}$s with no volume and no area, but
179. Length, of concrete type, for example, m.

For these we have that

180. the *intersection*, $\bigcap$, of two $\mathbb{EXTENT}$s is an $\mathbb{EXTENT}$ of possibly nil Volume,
181. the intersection, $\bigcap$, of two $\mathbb{SURFACE}$s may be either a possibly nil $\mathbb{SURFACE}$ or a possibly nil $\mathbb{LINE}$, or a combination of these.
182. the intersection, $\bigcap$, of two $\mathbb{LINE}$s may be either a possibly nil $\mathbb{LINE}$ or a $\mathbb{POINT}$.

Similarly we can define

183. the *union*, $\bigcup$, of two not-disjoint $\mathbb{EXTENT}$s,
184. the *union*, $\bigcup$, of two not-disjoint $\mathbb{SURFACE}$s,
185. the *union*, $\bigcup$, and of two not-disjoint $\mathbb{LINE}$s.

[5] `https://en.wikipedia.org/wiki/Technical_drawing`

nd:

186. the *[in]equality*, $\neq, =$, of pairs of $\mathbb{EXTENT}$, pairs of $\mathbb{SURFACE}$s, and pairs of $\mathbb{LINE}$s.

We invite the reader to first first express the signatures for these operations, then their pre-conditions, and finally, being courageous, appropriate fragments of axiom systems.

13.6 Metric Space

Definition 85 **Definite Space:** *By a* definite space *we shall understand a space with a definite metric* ■

Figure **13**.1 diagrams some mathematical models of space. We shall hint a just one of these spaces.

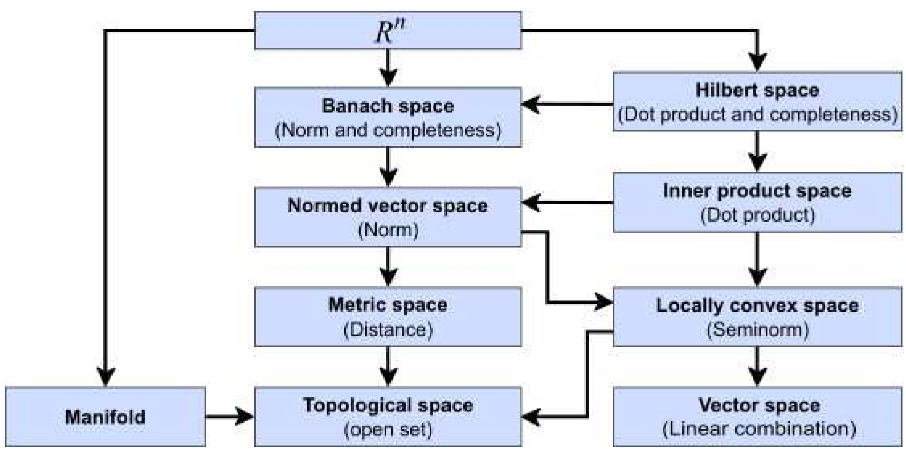


Figure 13.1: Variety of Abstract Spaces
An arrow from space A to space B implies that A is also a kind of B.

A metric space is an ordered pair (M, d) where M is a set and d is a metric on M, i.e., a function:

$$d : M \times M \to \textbf{Real}$$

such that for any $x, y, z \in M$, the following holds:

$$d(x, y) = 0 \equiv \ x = y \quad \text{identity of indiscernibles} \qquad (\mathbf{13}.1)$$

$$d(x, y) = d(y, x) \quad \text{symmetry} \qquad (\mathbf{13}.2)$$

$$d(x, z) \leq d(x, y) + d(y, z) \quad \text{sub-additivity or triangle inequality} \qquad (\mathbf{13}.3)$$

Given the above three axioms, we also have that $d(x, y) \geq 0$ for any $x, y \in M$. This is deduced as follows:

$$d(x, y) + d(y, x) \geq d(x, x) \quad \text{triangle inequality} \tag{13.4}$$

$$d(x, y) + d(y, x) \geq d(x, x) \quad \text{by symmetry} \tag{13.5}$$

$$2d(x, y) \geq 0 \quad \text{identity of indiscernibles} \tag{13.6}$$

$$d(x, y) \geq 0 \quad \text{non-negativity} \tag{13.7}$$

The function d is also called distance function or simply distance. Often, d is omitted and one just writes M for a metric space if it is clear from the context what metric is used.

13.7 Summary & Conclusion

Chapter 14

Geometry

Contents

14.1 Geometry

Characterisation: Geometry (from Ancient Greek $\gamma\epsilon\omega\epsilon\tau\rho\iota\alpha$ (`geōmetría, land measurement`) and from $\gamma\bar{\eta}$ (`gêm`, *earth, land*), and $\mu\epsilon\tau\rho\nu$ (`métron`, *a measure*) is a branch of mathematics concerned with properties of space such as the distance, shape, size, and relative position of figures.

14.1.1 Euclid's Elementa

https://en.wikipedia.org/wiki/Euclidean_geometry

14.1.1.1 Two Axiom Systems

14.1.1.1.1 Euclid: [1] Euclidean geometry is an axiomatic system, in which all theorems ("true statements") are derived from a small number of simple axioms. Until the advent of non-Euclidean geometry, these axioms were considered to be obviously true in the physical world, so that all the theorems would be equally true. However, Euclid's reasoning from assumptions to conclusions remains valid independent of their physical reality.

Near the beginning of the first book of the Elements, Euclid gives five postulates (axioms) for plane geometry, stated in terms of constructions (as translated by Thomas Heath):

Let the following be **postulated:**

- To draw a straight line from any point to any point.
- To produce (extend) a finite straight line continuously in a straight line.
- To describe a circle with any centre and distance (radius).
- That all right angles are equal to one another.
- [The parallel postulate]: That, if a straight line falling on two straight lines make the interior angles on the same side less than two right angles, the two straight lines, if produced indefinitely, meet on that side on which the angles are less than two right angles.

Although Euclid explicitly only asserts the existence of the constructed objects, in his reasoning he also implicitly assumes them to be unique.

The Elements also include the following five **common notions:**

- Things that are equal to the same thing are also equal to one another (the transitive property of a Euclidean relation).
- If equals are added to equals, then the wholes are equal (Addition property of equality).
- If equals are subtracted from equals, then the differences are equal (subtraction property of equality).
- Things that coincide with one another are equal to one another (reflexive property).
- The whole is greater than the part.

Modern scholars agree that Euclid's postulates do not provide the complete logical foundation that Euclid required for his presentation. Modern treatments use more extensive and complete sets of axioms.

[1] Euclid lived around year 300 BC in Greece

14.1.1.1.2 Birkhoff: In 1932, George David Birkhoff[2] created a set of four postulates of Euclidean geometry sometimes referred to as Birkhoff's axioms. These postulates are all based on basic geometry that can be experimentally verified with a scale and protractor. In a radical departure from the synthetic approach of Hilbert, Birkhoff was the first to build the foundations of geometry on the real number system. It is this powerful assumption that permits the small number of axioms in this system.

Postulates: Birkhoff uses four undefined terms: *point, ℓine, δistance* and *angle* ($\angle$). His postulates are:

- **Postulate I:** Postulate of *Line Measure.* The points $A, B, \ldots$ of any line can be put into $1:1$ correspondence with the real numbers x so that $\mid xB - xA \mid^3 = \delta(A, B)$ for all points A and B.

- **Postulate II:** Postulate of *Point-Line*. There is one and only one straight line, ℓ, that contains any two given distinct points P and Q.

- **Postulate III:** Postulate of *Angle Measure.* The rays $\{\ell, m, n, \ldots\}$ through any point O can be put into $1:1$ correspondence with the real numbers a ($\textbf{mod}\, 2$) so that if A and B are points (not equal to O) of ℓ and m, respectively, the difference $am - a\ell$ ($\textbf{mod}\, 2$) of the numbers associated with the lines ℓ and m is $\angle AOB$. Furthermore, if the point B on m varies continuously in a line r not containing the vertex O, the number am varies continuously also.

- **Postulate IV:** Postulate of *Similarity.* If in two triangles ABC and $A'B'C'$ and for some constant $k > 0, d(A', B') = kd(A, B), d(A', C') = k\delta(A, C)$ and $\angle B'A'C'^4 = \pm\angle BAC$, then $\delta(B', C') = k\delta(B, C), \angle C'B'A' = \pm\angle CBA$, and $\angle A'C'B' = \pm\angle ACB$.

---

[2]George David Birkhoff was an american mathematician (1884–1944)

[3]By xA and xB we mean the position on the line ℓ of points a and B, and by $|xB - xA\mid$ we mean the positive, i.e., numerical, *d*istance between points xB and yA

[4]By $\angle ABC$ we mean the angle between lines AB and AC

14.1.2

14.1.3

14.2 Trigonomety

14.2.1

14.2.2

14.2.3

14.3 Summary & Conclusion

Chapter 15

Time

Contents

a moving image of eternity;
the number of the movement in respect of the before and the after;
the life of the soul in movement as it passes
from one stage of act or experience to another;
a present of things past: memory, a present of things present: sight,
and a present of things future: expectations[1]

This thing all things devours:
Birds, beasts, trees, flowers;
Gnaws iron, bites steel,
Grinds hard stones to meal;
Slays king, ruins town,
And beats high mountain down.[2]

Concepts of time fascinate philosophers and scientists [42, 67, 70, 72–77, 79, 90] and [46].

15.1 Time Motivated Philosophically

Definition 86 Indefinite Time: *We motivate the abstract notion of time as follows.* [86, pp 159] "Two different states must necessarily be ascribed different incompatible predicates. But how can we ensure so? Only if states stand in an asymmetric relation to one

[1]Quoted from [2, Cambridge Dictionary of Philosophy]
[2]J.R.R. Tolkien, The Hobbit

another. This state relation is also transitive. So that is an indispensable property of any world. By a transcendental deduction we say that *primary entities exist in time*. So every possible world must exist in time" ■

Definition 87 Definite Time: By a definite time we shall understand an abstract representation of time such as for example *year*, *month*, *day*, *hour*, *minute*, *second*, etc. ■

15.2 $\mathbb{TIME}$

$\mathbb{TIME}$ may occur in endurant attribute descriptions and in perdurant descriptions.

15.2.1 $\mathbb{TIME}$ Type and Values

We shall not be concerned with any representation of time. That is, we leave it to the domain analyser cum describer to choose an own representation [46]. Similarly we shall not be concerned with any representation of time intervals.[3]

187. So there is an abstract type $\mathbb{T}ime$,
188. and an abstract type $\mathbb{TI}$: $\mathbb{T}ime\mathbb{I}nterval$.
189. There is no $\mathbb{T}ime$ origin, but there is a "zero" $\mathbb{TI}$me interval.
190. One can add (subtract) a time interval to (from) a time and obtain a time.
191. One can add and subtract two time intervals and obtain a time interval – with subtraction respecting that the subtrahend is smaller than or equal to the minuend.
192. One can subtract a time from another time obtaining a time interval respecting that the subtrahend is smaller than or equal to the minuend.
193. One can multiply a time interval with a real and obtain a time interval.
194. One can compare two times and two time intervals.

type
187 $\mathbb{T}$
188 $\mathbb{TI}$
value
189 $\mathbf{0}$:$\mathbb{TI}$
190 $+,-$: $\mathbb{T} \times \mathbb{TI} \to \mathbb{T}$
191 $+,-$: $\mathbb{TI} \times \mathbb{TI} \xrightarrow{\sim} \mathbb{TI}$
192 $-$: $\mathbb{T} \times \mathbb{T} \to \mathbb{TI}$
193 $*$: $\mathbb{TI} \times$ **Real** $\to \mathbb{TI}$
194 $<,\leq,=,\neq,\geq,>$: $\mathbb{T} \times \mathbb{T} \to$ **Bool**
194 $<,\leq,=,\neq,\geq,>$: $\mathbb{TI} \times \mathbb{TI} \to$ **Bool**
axiom
190 $\forall$ t:$\mathbb{T}$ • t+$\mathbf{0}$ = t

[3]– but point out, that although a definite time interval may be referred to by number of years, number of days (less than 365), number of hours (less than 24), number of minutes (less than 60)number of seconds (less than 60), et cetera, this is not a time, but a time interval.

15.2.2 $\mathbb{TIME}$ Observer

195. We define the signature of the meta-physical time observer.

value
195 **record_**$\mathbb{TIME}$: **Unit** $\rightarrow \mathbb{T}$

The time recorder applies to nothing and yields a time. **record_**$\mathbb{TIME}$() occur in action, event and behavioural descriptions.

• • •

Modern models of time, by mathematicians and physicists evolve around spacetime[4] We shall not be concerned with this notion of time.

Models of time related to domain modeling and programming differs from those of mathematicians and physicists in focusing on divergence and convergence, zero (Zenon) time and interleaving time [93] are relevant in studies of real-time, typically distributed computing systems. We shall also not be concerned with this notion of time. But we shall refer the able reader to two seminal works on time in respect of modeling and programming: [94, *Zhou Chao Chen & Michael Reichhardt Hansen*] and [63, *Leslie Lamport*].

15.3 Axiom Systems

15.3.1 Johan van Benthem

The following is taken from Johan van Benthem [90]:

196. Let P be a point structure (for example, a set).

197. Think of time as a continuum;

198. the following axioms characterise ordering ($<$, $=$, $>$) relations between (i.e., aspects of) time points.

199. The axioms listed below are not thought of as an axiom system, that is, as a set of independent axioms all claimed to hold for the time concept, which we are encircling.

200. Instead van Benthem offers the individual axioms as possible "blocks" from which we can then "build" our own time system — one that suits the application at hand, while also fitting our intuition.

201. Time is transitive: If $p<p'$ and $p'<p''$ then $p<p''$.

[4]The concept of **Spacetime** was first "announced" by Hermann Minkowski, 1907–08 – based on work by Henri Poincaré, 1905–06, https://en.wikisource.org/wiki/Translation: The_Fundamental_Equations_for_Electromagnetic_Processes_in_Moving_Bodies

202. Time may not loop, that is, is not reflexive: $p \not< p$.

203. Linear time can be defined: Either one time comes before, or is equal to, or comes after another time.

204. Time can be left-linear, i.e., linear "to the left" of a given time.

205. One could designate a time axis as beginning at some time, that is, having no predecessor times.

206. And one can designate a time axis as ending at some time, that is, having no successor times.

207. General, past and future successors (predecessors, respectively successors in daily talk) can be defined.

208. Time can be dense: Given any two times one can always find a time between them.

209. Discrete time can be defined.

15.3.1.1 A Continuum Theory of Time

198. $[\ \mathsf{LIN: Linearity}\ ]\ \forall\ p,p':P \bullet (p{=}p' \vee p{<}p' \vee p{>}p')$
199. $[\ \mathsf{L{-}LIN: Left\ Linearity}\ ]$
199. $\quad \forall\ p,p',p'':P \bullet (p'{<}p \wedge p''{<}p) \Rightarrow (p'{<}p'' \vee p'{=}p'' \vee p''{<}p')$
200. $[\ \mathsf{BEG: Beginning}\ ]\ \exists\ p:P \bullet \sim\exists\ p':P \bullet p'{<}p$
201. $[\ \mathsf{END: Ending}\ ]\ \exists\ p:P \bullet \sim\exists\ p':P \bullet p{<}p'$
202. $[\ \mathsf{SUCC: Successor}\ ]$
202. $\quad [\ \mathsf{PAST: Predecessors}\ ]\ \forall\ p:P,\exists\ p':P \bullet p'{<}p$
202. $\quad [\ \mathsf{FUTURE: Successor}\ ]\ \forall\ p:P,\exists\ p':P \bullet p{<}p'$
203. $[\ \mathsf{DENS: Dense}\ ]\ \forall\ p,p':P\ (p{<}p' \Rightarrow \exists\ p'':P \bullet p{<}p''{<}p')$
204. $[\ \mathsf{CDENS: Converse\ Dense}\ ] \equiv [\ \mathsf{TRANS: Transitivity}\ ]$
204. $\quad \forall\ p,p':P\ (\exists\ p'':P \bullet p{<}p''{<}p' \Rightarrow p{<}p')$
205. $[\ \mathsf{DISC: Discrete}\ ]$
205. $\quad \forall\ p,p':P \bullet (p{<}p' \Rightarrow \exists\ p'':P \bullet (p{<}p'' \wedge \sim\exists\ p''':P \bullet (p{<}p'''{<}p''))) \wedge$
205. $\quad \forall\ p,p':P \bullet (p{<}p' \Rightarrow \exists\ p'':P \bullet (p''{<}p' \wedge \sim\exists\ p''':P \bullet (p''{<}p'''{<}p')))$
206. $[\ \mathsf{TRANS: Transitivity}\ ]\ \forall\ p,p',p'':P \bullet p < p' < p'' \Rightarrow p < p''$
207. $[\ \mathsf{IRREF: Irreflexitivity}\ ]\ \forall\ p:P \bullet p \not< p$
208. $[\ \mathsf{LIN: Linearity}\ ]\ \forall\ p,p':P \bullet (p{=}p' \vee p{<}p' \vee p{>}p')$
209. $[\ \mathsf{L{-}LIN: Left\ Linearity}\ ]$
207. $\quad \forall\ p,p',p'':P \bullet (p'{<}p \wedge p''{<}p) \Rightarrow (p'{<}p'' \vee p'{=}p'' \vee p''{<}p')$
207. $[\ \mathsf{BEG: Beginning}\ ]\ \exists\ p:P \bullet \sim\exists\ p':P \bullet p'{<}p$
207. $[\ \mathsf{END: Ending}\ ]\ \exists\ p:P \bullet \sim\exists\ p':P \bullet p{<}p'$

- A *strict partial order*, SPO, is a *point structure* satisfying TRANS and IRREF.

- TRANS, IRREF and SUCC imply *infinite models.*
- TRANS and SUCC may have *finite, "looping time" models.*

15.3.2 Wayne Blizzard

Wayne D. Blizard [35, 1980] relates abstracted entities to spatial points and time.

We shall present an axiom system [35, Wayne D. Blizard, 1980] which relate abstracted entities to spatial points and time. Let $A, B, \ldots$ stand for entitites, $p, q, \ldots$ for spatial points, and t, τ for times. 0 designates a first, a begin time. Let t' stand for the discrete time successor of time t. Let $N(p,q)$ express that p and q are spatial neighbours. Let $=$ be an overloaded equality operator applicable, pairwise to entities, spatial locations and times, respectively. A_p^t expresses that entity A is at location p at time t. The axioms — where we omit (obvious) typings (of A, B, P, Q, and T): $'$ designates the time successsor function: t'.

15.3.2.1 A Theory of Time–Space

$$
\begin{array}{rlrcll}
(I) & & \forall A \forall t \exists p & : & A_p^t & \\
(II) & & (A_p^t \wedge A_q^t) & \supset & p = q & \\
(III) & & (A_p^t \wedge B_p^t) & \supset & A = B & \\
(IV) & & (A_p^t \wedge A_p^{t'}) & \supset & t = t' & \\
(V & .i) & \forall p, q & : & N(p,q) \supset p \neq q & \text{Irreflexivity} \\
(V & .ii) & \forall p, q & : & N(p,q) = N(q,p) & \text{Symmetry} \\
(V & .iii) & \forall p \exists q, r & : & N(p,q) \wedge N(p,r) \wedge q \neq r & \text{No isolated locations} \\
(VI & .i) & \forall t & : & t \neq t' & \\
(VI & .ii) & \forall t & : & t' \neq 0 & \\
(VI & .iii) & \forall t & : & t \neq 0 \supset \exists \tau : t = \tau' & \\
(VI & .iv) & \forall t, \tau & : & \tau' = t' \supset \tau = t & \\
(VII) & & A_p^t \wedge A_q^{t'} & \supset & N(p,q) & \\
(VIII) & & A_p^t \wedge B_q^t \wedge N(p,q) & \supset & \sim (A_q^{t'} \wedge B_p^{t'}) &
\end{array}
$$

(II–IV,VII–VIII): The axioms are universally 'closed'; that is: We have omitted the usual $\forall A, B, p, q, t$s.

(I): For every entity, A, and every time, t, there is a location, p, at which A is located at time t.

(II): An entity cannot be in two locations at the same time.

(III): Two distinct entities cannot be at the same location at the same time.

(IV): Entities always move: An entity cannot be at the same location at different times. *This is more like a conjecture: Could be questioned.*

(V): These three axioms define N.

(V.i): Same as $\forall p : \sim N(p,p)$. "Being a neighbour of", is the same as "being distinct from".

(V.ii): If p is a neighbour of q, then q is a neighbour of p.

(V.iii): Every location has at least two distinct neighbours.

(VI): The next four axioms determine the time successor function $'$.

(VI.i): A time is always distinct from its successor: time cannot rest. There are no time fix points.

(VI.ii): Any time successor is distinct from the begin time. Time 0 has no predecessor.

(VI iii): Every non–begin time has an immediate predecessor.

(VI.iv): The time successor function $'$ is a one–to–one (i.e., a bijection) function.

(VII): The *continuous path axiom:* If entity A is at location p at time t, and it is at location q in the immediate next time (t'), then p and q are neighbours.

(VIII): *No "switching":* If entities A and B occupy neighbouring locations at time t them it is not possible for A and B to have switched locations at the next time (t').

Except for Axiom (IV) the system applies both to systems of entities that "sometimes" rests, i.e., do not move. These entities are spatial and occupy at least a point in space. If some entities "occupy more" space volume than others, then we interpret, in a suitable manner, the notion of the point space P (etc.). We do not show so here.

Part IV

Structured Clauses

Chapter 16

Structured Expressions

Contents

16.1 Patterns

16.2 The let ... = ... in ... end Expression

16.2.1 General

16.2.2 Function Definitions

16.3 The Conditional Clauses

16.3.1 The if ... then ... else ... end Expression

16.3.2 The cases ... of ... → .., ..., ... → ... end Expression

16.3.3 The McCarthyExpression

16.4 ...

16.5 ...

16.6 Summary

Part V

Elaboration Order

Chapter 17

Sequentiality

$\boxed{\text{to be written}}$

17.1 A State Concept

$\boxed{\text{to be written}}$

17.2 Imperative Programming

$\boxed{\text{to be written}}$

17.3 Summary

$\boxed{\text{to be written}}$

Chapter 18

Concurrency

$\boxed{\text{to be written}}$

18.1 Traces

$\boxed{\text{to be written}}$

18.2 CSP: Communicating Sequential Processes

$\boxed{\text{to be written}}$

18.3 From Endurants to Perdurants

$\boxed{\text{to be written}}$

Example 72 **From Road Net Endurants to Automobile Behaviour:** ■

18.4 Petri Nets

$\boxed{\text{to be written}}$

18.5 Other Forms of Concurrent Specifications

$\boxed{\text{to be written}}$

18.6 Summary

$\boxed{\text{to be written}}$

Part VI

Algorithms

Chapter 19

Applicative Algorithms

Chapter 20

Imperative Algorithms

Chapter 21

Concurrent Algorithms

Part VII

Programming Paradigms

Chapter 22

Functional Programming Languages

Contents

https://en.wikipedia.org/wiki/Functional_programming

22.1 Introduction

22.2 Church's λ-Calculus

22.3 Landin's AE

22.4 Milner's LCF and ML

22.5 Haskell

22.6 F#

22.7 ...

22.8 ......

22.9 Summary and Conclusion

Chapter 23

Imperative Programming Languages

Contents

23.1 Introduction

23.2 Turing's Machines

23.3 Machine Languages

23.4 Assembly

23.5 Backus' Fortran

23.6 The Committee Algol 60

23.7 van Wijgaarden's Algol 68

23.8 Wirth's Algol-derivatices

23.8.1 Algol W

23.8.2 Pascal

23.8.3 Modula

23.8.4 Oberon

23.8.5 The "Impossibly Large" Languages

23.8.6 CHILL

23.8.7 Ada

23.9 Java

23.10 ...

23.11 Python

23.12 Summary and Conclusion

Chapter 24

Parallel Programming Languages

Chapter 25

Logic Programming Languages

Contents

25.1 Introduction

https://en.wikipedia.org/wiki/Logic_programming

25.2 Colmerauer's Prolog

25.3 Constraint Logic Programming:

25.4 ...

25.5 ......

25.6 Summary and Conclusion

Part VIII

Modeling Paradigms

Chapter 26

Domain Modeling

Chapter 27

Petri Modeling

Chapter 28

Algebraic Modeling

Part IX

Semiotics

Chapter 29

Syntax

Chapter 30

Semantics

Chapter 31

Pragmatics

Part X

The Triptych Dogma

Chapter 32

Domains, II

Chapter 33

Requirements

Chapter 34

Software

Part XI

Domains II

Chapter 35

Mathematics

Contents

$\boxed{\text{to be written}}$

35.1 Review

35.2 Logic

35.3 Sets

35.4 Number Theory

35.5 Cartesians

35.6 Graphs

35.7 ...

35.8 ...

35.9 Summary and Conclusion

Chapter 36

Natural Sciences

Contents

$\boxed{\text{to be written}}$

36.1 Overview

36.2 Physics

36.2.1 Mechanics

36.2.1.1 Mechanical Parts

36.2.1.2 Newton's Laws

36.2.1.3 Kinematics

36.2.1.4 Discussion

36.2.2 Electricity

36.2.3 Fluid Mechanics

36.2.4 Thermodynamics

36.3 Botanics

36.4 Zoology

36.5 Biology

36.6 Geography

36.7 Geology

36.8 Metereology

36.8.1 Weather

36.8.2 Statics

36.8.3 Dynamics

36.9 Other

36.10 Summary and Conclusion

Chapter 37

Transport

Contents

$\boxed{\text{to be written}}$

37.1 Nets

$\boxed{\text{to be written}}$

37.2 Road

Example 73 Road Transport, II:

We present the types of a simple **domain model** of road transport.

The External Qualities of Parts:

210. There are road transports, RT.

211. From a road transport we can observe an aggregate of a **road net**, RN, and an **aggregate of automobiles**, AA.

212. From an aggregate of a road net we can observe an **aggregate of street intersections**, we shall call them **hubs**, AH, and **street segments**, i.e., **links**, as we

shall call them, AL, [directly, immediately] between two hubs.

213. From an aggregate of automobiles we can observe a **set of automobiles**, As.

214. From an aggregate of hubs we can observe a **set of hubs** Hs.

215. From an aggregate of links we can observe a **set of links** Ls.

216. **Automobiles** A, **hubs** H, and **links** L and are here considered **atomic**, i.e., consists of not further sub-parts.

type
210. RT
211. RN
211. AA
212. AH, AL
213. As = A-**set**
214. Hs = H-**set**
215. Ls = L-**set**
216. A, H, L

value
211. obs_RN: RT $\rightarrow$ RN
211. obs_AA: RT $\rightarrow$ AA
212. obs_AH: RN $\rightarrow$ AH
212. obs_AL: RN $\rightarrow$ AL
213. obs_As: AA $\rightarrow$ As
214. obs_Hs: AH $\rightarrow$ Hs
215. obs_Ls: AL $\rightarrow$ Ls

The Internal Qualities of Parts: Unique Identifiers, Mereologies and Attributes

Automobiles:

217. **Automobiles** have **unique identifiers**,

218. are **mereologically** related to a subset of all hubs and links, and

219. have **attributes** of position, APos, on the road net [programmable], velocity, AVel [programmable], history, AHis, of the times, $\mathbb{TIME}$, they left and entered hubs and links and the road net, and entered the road net, links and hubs.

Hubs:

220. **Hubs** have **unique identifiers**,

221. are **mereologically** related to the one[1] or two links it connects (i.e., upon which it is incident), and

222. have **attributes** of current signal state, Σ [programmable], signal state space, Ω [static], and **hub history**, HHis [programmable], i.e., the times automobiles left and entered the hub.

Links:

223. **Links** have unique **identifiers**,

[1] One if the link "loops back" to the hub from which it emanates

224. are **mereologically** related to the one or two hubs upon which they are incident, and

225. have **attributes** of **length**, LEN [static] and **link history**, LHis [programmable], i.e., the times automobiles left and entered the link.

type

Unique Identification:

217. AI
220. HI
223. LI

value

217. uid_A: A $\rightarrow$ AI
220. uid_H: H $\rightarrow$ HI
223. uid_L: L $\rightarrow$ LI

Mereology:

type

218. AM = (HI|LI)-**set**
221. HM = LI-**set**
224. LM = HI-**set**
224. [**axiom** $\forall$ lm:LM•1$\leq$**card** lm$\leq$2]

value

218. mereo_A: A $\rightarrow$ AM
221. mereo_H: H $\rightarrow$ HM
224. mereo_L: L $\rightarrow$ LM

Attributes:

type

219. APos, AVel
219. AHis = ($\mathbb{TIME}\times$(HI|LI))*
222. HΣ = LI-**set**
222. [**axiom** $\forall$ h:H•**card** 1$\leq$attr_HΣ(h)$\leq$2]
222. HΩ = HΣ-**set**
222. [**axiom** $\forall$ h:H•attr_HΣ(h)$\in$attr_HΩ(h)]
222. HHis = ($\mathbb{TIME}\times$AI)*
225. LEN
222. LHis = ($\mathbb{TIME}\times$AI)*

value

219. attr_APos: A $\rightarrow$ APos
219. attr_AVel: A $\rightarrow$ AVel
219. attr_AHis: A $\rightarrow$ AHis
222. attr_HΣ: H $\rightarrow$ HΣ
222. attr_HΩ: H $\rightarrow$ HΩ
225. attr_LEN: L $\rightarrow$ LEN
225. attr_LHis: L $\rightarrow$ LHis

Intentions and Intentional Pull: We narrate, but do not formalize:

Intentions:

226. The intentions of road transport is
   (a) for automobiles to drive on roads: entering and leaving hubs and links, driving around hubs and along links, sometimes stopping, and
   (b) for hubs and links to accommodate automobiles: letting them enter and leave, drive around or along.

Intentional Pull:

227. For any automobile a in the road net
   (a) if at some time τ it is leaving a hub h or a link ℓ,
   (b) then that hub or link has recorded that event.

228. For any hub h [link ℓ] of the road net

(a) if at some time τ it observes an automobile a entering (leaving) that hub [or link]

(b) then that automobile a has recorded that corresponding event.

• • •

These are the main characteristics of **solid** road transport **endurants**, i.e., **parts**. Other characteristics, such as the **perdurants**, i.e., *behaviours*, **action**s and **events** will be exemplified later[2] ■

37.3 Rail

to be written

37.4 Air

to be written

37.5 Sea

to be written

37.6 Summary and Conclusion

to be written

[2] Editorial Note: **Remember to develop these examples and insert reference.**

Chapter 38

Documents

Contents

---

to be written

38.1 Introduction

38.1.1 Physical Documents

38.1.2 Abstract Documents

38.2 Documents in General

38.2.1 Unique Identification

38.2.2 Mereology

38.2.2.1 Cross Referencing

38.2.2.2 Dangling References

38.2.3 Attributes

38.2.3.1 Text

38.2.3.2 Owners

38.2.3.3 Access Rights

38.2.3.3.1 Readers,

38.2.3.3.2 Editors.

38.2.3.3.3 Copiers.

38.2.3.3.4 Destructors.

38.2.3.4 History

38.2.3.5

38.2.4 Operations

38.2.4.1 Creation

38.2.4.2 Editing

38.2.4.3 Reading

38.2.4.4 Copying

38.2.4.5 Searching

38.2.4.6 Destruction

38.3 Formal Documents

38.3.1 Certificates

38.3.2 Contracts

to be written

38.4 Summary and Conclusion

Chapter 39

Industry

Contents

39.1 Introduction

39.2 Finance

39.2.1 Monies

39.2.1.1 Notes and Coins

39.2.1.2 Currencies

39.2.1.3 Debt Certificates

39.2.2 Banks

39.2.3 Credit Cards

39.2.4 Insurance

39.2.5 Other

39.3 Pipe Lines

39.3.1 Water

39.3.2 Oil and Gas

39.3.3 Gravel

39.3.4 Etcetera

39.3.5 Common Model

39.4 Manufacturing

39.4.1 ...

39.4.2 ... ...

39.5 ... ...

39.6 Summary and Conclusion

Chapter 40

Public Services

Contents

40.1 Health Care

40.2 Libraries

40.3 ...

40.4 ......

40.5 ....

40.6 Summary and Conclusion

Chapter 41

Services

Contents

41.1 Retailing

41.2 Hotels

41.3 Restaurants, Cafés and Bars

41.4 Construction

41.4.1 Builder

41.4.2 Painter

41.4.3 Carpenter

41.5 Summary and Conclusion

Chapter 42

Management

Contents

to be written

42.1 Bookkeeping

42.1.1 Accounts

42.1.2 Accounting

42.1.3 Profit and Loss

42.2 Projects

42.2.1 Resources

42.2.2 Plans

42.2.3 Planning

42.2.4 Monitoring

42.2.5 Control

42.2.6 Resource Allocation

42.3 Summary and Conclusion

Chapter 43

Ontology & Taxonomies

Contents

Motivation: ...

We remind the reader of our definitions of the terms: *ontology* and *taxonomy*, Sect. **0.8** on page 11.

In this book we have relied on the *domain analysis & description ontology*, Fig. **1**.1 of Sect. **1.2** and on the *attributes ontology*, Fig. ?? on page ?? of Sect. ??. And we have shown *a road transport taxonomy*, Fig. ?? on page ?? Sect. ??.

Dels for External Qs, bde O og T, f.eks. T for road transport

og for Attributes, bde O og T, f.eks. T for automobile attrs.: Static, monitoravle herunder bid-davle og programmellet.

Ontology: one for all domain analysis and description dets., i.e., shared, general.

Taxonomy: one for each domain model, i.e., particular.

43.1 Review of Domain Modeling Ontology and Taxonomies

to be written

43.2 Carl von Linné's Plant Taxonomy

- https://botanicalsociety.org.za/the-science-of-names-an-introduction-to-plant-taxon
- https://www.botanicalartandartists.com/plant-evolution-and-taxonomy.html
- https://en.m.wikipedia.org/wiki/Plant_taxonomy

Plant taxonomy is the science that finds, identifies, describes, classifies, and names plants. It is one of the main branches of taxonomy (the science that finds, describes, classifies, and names living things).

Plant identification is a determination of the identity of an unknown plant by comparison with previously collected specimens or with the aid of books or identification manuals. The process of identification connects the specimen with a published name. Once a plant specimen has been identified, its name and properties are known.

Plant classification is the placing of known plants into groups or categories to show some relationship. Scientific classification follows a system of rules that standardizes the results, and groups successive categories into a hierarchy. For example, the family to which the lilies belong is classified as follows:

- Kingdom: Plantae
- Division: Magnoliophyta
- Class: Liliopsida
- Order: Liliales
- Family: Liliaceae

•

Part XII

Formal Bases

Chapter 44

Mathematical Logic

Chapter 45

Axiom Systems

Chapter 46

Algebras

In this chapter we shall elaborate on the mathematical concept of **algebras**.

Motivation: Algebras

Computer science has developed a number of interpretations, i.e., uses, of the mathematical concept of algebras [3,6,7,40,53,64,80]. Most notably this has found a beautiful form in [81, *Sanelle & Tarlecki*].

$\boxed{\text{to be written}}$

In our methodology for analysing and describing domains it seems that we can identify a number of algebras: an $\mathcal{A}$lgebra, $\mathcal{A}_x$, of e$x$ternal qualities of endurants, an $\mathcal{A}$lgebra, $\mathcal{A}_u$, of the internal qualities of $u$nique identifiers of endurant, an $\mathcal{A}$lgebra, $\mathcal{A}_m$, of the internal qualities of $m$ereologies of endurants, an $\mathcal{A}$lgebra, $\mathcal{A}_a$, of the internal qualities of $a$ttributes of endurants, and possibly many more!

$\boxed{\text{much more to come}}$

Chapter 47

Recursive Function Theory

Chapter 48

Verification

Part XIII

Closing

Chapter 49

Closing

Chapter 50

Bibliography

Contents

50.1 Bibliographical Notes

50.2 References

[1] .

[2] Rober Audi. *The Cambridge Dictionary of Philosophy*. Cambridge University Press, The Pitt Building, Trumpington Street, Cambridge CB2 1RP, England, 1995.

[3] M. Barr and G. Wells. *Category Theory for Computing Science*. Prentice-Hall, 1990.

[4] Claude Berge. *Théorie des Graphes et ses Applications*. Collection Universitaire de Mathematiques. Dunod, Paris, 1958. See [5].

[5] Claude Berge. *Graphs*, volume 6 of *Mathematical Library*. North-Holland Publ. Co., second revised edition of part 1 of the 1973 english version edition, 1985. See [4].

[6] Garrett Birkhoff. On the structure of abstract algebras. *Proceedings of the Cambridge Philosophical Society*, 31:433–454, 1935.

[7] Garrett Birkhoff. *Lattice Theory*, volume 25 of *American Mathematical Societ Colloquium Publications*. American Mathematical Society, New York, N.Y., third edition edition, 1967.

[8] Dines Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. See [11, 14].

[9] Dines Bjørner. *Software Engineering, Vol. 2: Specification of Systems and Languages*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. Chapters 12–14 are primarily authored by Christian Krog Madsen. See [12, 15].

[10] Dines Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. See [13, 16].

[11] Dines Bjørner. *Software Engineering, Vol. 1: Abstraction and Modelling*. Qinghua University Press, 2008.

[12] Dines Bjørner. *Software Engineering, Vol. 2: Specification of Systems and Languages*. Qinghua University Press, 2008.

[13] Dines Bjørner. *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Qinghua University Press, 2008.

[14] Dines Bjørner. **Chinese:** *Software Engineering, Vol. 1: Abstraction and Modelling*. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.

[15] Dines Bjørner. **Chinese:** *Software Engineering, Vol. 2: Specification of Systems and Languages*. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.

[16] Dines Bjørner. **Chinese:** *Software Engineering, Vol. 3: Domains, Requirements and Software Design*. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.

[17] Dines Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics, Part I of II: The Engineering Part*. *Kibernetika i sistemny analiz*, 2(4):100–116, May 2010.

[18] Dines Bjørner. Domain Science & Engineering – *From Computer Science to The Sciences of Informatics Part II of II: The Science Part*. *Kibernetika i sistemny analiz*, 2(3):100–120, June 2011.

[19] Dines Bjørner. Pipelines – a Domain `www.imm.dtu.dk/~dibj/pipe-p.pdf`. Experimental Research Report 2013-2, DTU Compute and Fredsvej 11, DK-2840 Holte, Denmark, Spring 2013.

[20] Dines Bjørner. Manifest Domains: Analysis & Description `www.imm.dtu.dk/ dibj/-2015/faoc/faoc-bjorner.pdf`. *Formal Aspects of Computing*, 29(2):175–225, March 2017. Online: 26 July 2016.

[21] Dines Bjørner. Container Terminals. `www.imm.dtu.dk/ dibj/2018/yangshan/-maersk-pa.pdf`. Technical report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, September 2018. An incomplete draft report; currently 60+ pages.

[22] Dines Bjørner. Domain Analysis & Description – Principles, Techniques and Modelling Languages. `www.imm.dtu.dk/~dibj/2018/tosem/Bjorner-TOSEM.pdf`. *ACM Trans. on Software Engineering and Methodology*, 28(2), April 2019. 68 pages.

[23] Dines Bjørner. *Domain Science & Engineering – A Foundation for Software Development*. EATCS Monographs in Theoretical Computer Science. Springer, 2021. A revised version of this book is [29].

[24] Dines Bjørner. Rivers and Canals. `www.imm.dtu.dk/~dibj/2021/Graphs/Rivers--and-Canals.pdf`. Technical Report, Technical University of Denmark, Fredsvej 11, DK-2840 Holte, Denmark, March 2021.

[25] Dines Bjørner. Documents: A Basis for Government. In *United Natonans Inst., Festschrift for Tomas Janowski and Elsa Estevez*. Guimaraes, Portgal, October, `www.imm.dtu.dk/~dibj/2022/janowski/docs.pdf`, 2022.

[26] Dines Bjørner. Domain Modelling. In Jonathan Bowen et al., editor, *Theories of Programming and Formal Methods: Essays Dedicated to Jifeng He on the Occasion of His 80th Birthday*, Lecture Notes in Computer Science, Festschrift. Spriger, August 2023.

[27] Dines Bjørner. Domain Modelling. Technical University of Denmark. Revised edition of [29]. xii+208 pages. `http://www.imm.dtu.dk/ dibj/2023/DomainModelling/DomainModelling.pdf`, June 2023.

[28] Dines Bjørner. Domain Modelling – A Primer. A short version of [29]. xii+202 pages[1], May 2023.

[29] Dines Bjørner. Domain Science & Engineering – A Foundation for Software Development. Revised edition of [23]. xii+346 pages[2], January 2023.

[30] Dines Bjørner. Domain Science: Facets. *To be submitted*, October 2023. Institute of Mathematics and Computer Science. Technical University of Denmark.

[31] Dines Bjørner. Domain Science: Interpretations – Simulators, Monitors and Controllers. *To be submitted*, October 2023. Institute of Mathematics and Computer Science. Technical University of Denmark.

[32] Dines Bjørner. Domain Science: Ontology and Taxonomies. *To be submitted*, October 2023. Institute of Mathematics and Computer Science. Technical University of Denmark.

[33] Dines Bjørner. Informatics for Beginners. Technical University of Denmark, 18 August 2023.

[34] Dines Bjørner. Pipelines: A Domain Science & Engineering Description. In *FSEN 2023: Fundamentals of Software Engineering, Teheran, Iran, May 35*. `www.imm.dtu.dk/~dibj/2023/tehran/tehran.pdf`, 2023.

[1] This book is currently being trasnlated into Chinese by Dr. Yang ShaoFa, IoSCAS, Beijing and into Russian by Dr. Mikhail Chupilko, ISP/RAS, Moscow

[2] Due to copyright reasons no URL is given to this document's possible Internet location. A primer version, omitting certain chapters, is [28]

[35] Wayne D. Blizard. A Formal Theory of Objects, Space and Time. *The Journal of Symbolic Logic*, 55(1):74–89, March 1990.

[36] M. Bunge. *Treatise on Basic Philosophy: Ontology I: The Furniture of the World*, volume 3. Reidel, Boston, Mass., USA, 1977.

[37] M. Bunge. *Treatise on Basic Philosophy: Ontology II: A World of Systems*, volume 4. Reidel, Boston, Mass., USA, 1979.

[38] Rudolf Carnap. *Introduction to Semantics*. Harvard Univ. Press, Cambridge, Mass., 1942.

[39] Roberto Casati and Achille C. Varzi. *Parts and Places: the structures of spatial representation*. MIT Press, 1999.

[40] P. Cohn. *Universal Algebra*. Reidel, 1981.

[41] O.-J. Dahl, E.W. Dijkstra, and Charles Anthony Richard Hoare. *Structured Programming*. Academic Press, 1972.

[42] David John Farmer. *Being in time: The nature of time in light of McTaggart's paradox*. University Press of America, Lanham, Maryland, 1990. 223 pages.

[43] W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra, editors. *Beauty is Our Business*, Texts and Monographs in Computer Science, New York, NY, USA, 1990. Springer. A Birthday Salute to Edsger W. Dijkstra.

[44] A. Fraenkel. *Set Theory*. North-Holland Publ.Co., Amsterdam, 1961.

[45] Abraham Fraenkel, Yehoshua Bar-Hillel, and Azriel Levy. *Foundations of Set Theory*. Elsevier Science Publ. Co., Amsterdam, The Netherlands, 2nd revised edition, 1 Jan 1973.

[46] Carlo A. Furia, Dino Mandrioli, Angelo Morzenti, and Matteo Rossi. *Modeling Time in Computing*. Monographs in Theoretical Computer Science. Springer, 2012.

[47] Chris W. George, Peter Haff, Klaus Havelund, Anne Elisabeth Haxthausen, Robert Milne, Claus Bendix Nielsen, Søren Prehn, and Kim Ritter Wagner. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992.

[48] Chris W. George, Anne Elisabeth Haxthausen, Steven Hughes, Robert Milne, Søren Prehn, and Jan Storbank Pedersen. *The RAISE Development Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.

[49] G. H. Hardy, Edward M. Wright, and John Silvermann. *An Introduction to the Theory of Numbers*. Oxford University Press, England, 6th edition edition, 2008. Editor: Roger Heath Brown.

[50] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[51] David Harel and Rami Marelly. *Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.

[52] Frank Harrary. *Graph Theory*. Addison Wesley Publishing Co., 1972.

[53] H. Herrlich and G.E. Strecker. *Category Theory*. Allyn and Bacon, Boston, 1973.

[54] Charles Anthony Richard Hoare. Notes on Data Structuring. In *[41]*, pages 83–174, 1972.

[55] Charles Anthony Richard Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985. Published electronically: `usingcsp.com/cspbook.pdf` (2004).

[56] IEEE Computer Society. IEEE–STD 610.12-1990: Standard Glossary of Software Engineering Terminology. Technical report, IEEE, IEEE Headquarters Office, 1730 Massachusetts Avenue, N.W., Washington, DC 20036-1992, USA. Phone: +1-202-371-0101, FAX: +1-202-728-9614, 1990.

[57] ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992, 1996, 1999.

[58] Michael A. Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. ACM Press. Addison-Wesley, Reading, England, 1995.

[59] Michael A. Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. ACM Press. Addison-Wesley Publishing Company, Wokingham, nr. Reading, England; E-mail: ipc@awpub.add-wes.co.uk, 1995. ISBN 0-201-87712-0; xiv + 228 pages.

[60] J.W. Backus and F.L. Bauer and J.Green and C. Katz and J. McCarthy and P. Naur and A.J. Perlis and H. Rutishauser and K. Samelson and B. Vauquois and J.H. Wegstein and A. van Wijngaarden and M. Woodger. Revised Report on the Algorithmic Language Algol 60 – edited by P. Naur. *The Computer Journal*, 5(4):349367, 1963.

[61] Andrew Kennedy. *Programming languages and dimensions*. PhD thesis, University of Cambridge, Computer Laboratory, April 1996. 149 pages: `cl.cam.ac.uk/techreports/UCAM-CL-TR-391.pdf`. Technical report UCAM-CL-TR-391, ISSN 1476-298.

[62] Stephen Cole Kleene. *Mathematical Logic*. Dover Publications, Dover Edition, December 1, 2002. Originally published in 1967 by John Wiley & Sons, Publ., New York, NY, USA.

[63] Leslie Lamport. *Specifying Systems*. Addison–Wesley, Boston, Mass., USA, 2002.

[64] Saunders Mac Lane. *Categories For The Working Mathematician*. Springer, 1971.

[65] W. Little, H.W. Fowler, J. Coulson, and C.T. Onions. *The Shorter Oxford English Dictionary on Historical Principles*. Clarendon Press, Oxford, England, 1973, 1987. Two vols.

[66] Michael J. Loux. *Metaphysics, a Contemporary Introduction*. Routledge Contemporary Introductions to Philosophy. Routledge, London and New York, 1998 (2nd ed., 2020).

[67] J. M. E. McTaggart. The Unreality of Time. *Mind*, 18(68):457–84, October 1908. New Series. See also: [70].

[68] Jacob Mey. *Pragmatics: An Introduction*. Blackwell Publishers, 13 January, 2001. Paperback.

[69] Charles Sanders Peirce. *Pragmatism as a Principle and Method of right thinking: The 1903 Harvard Lectures on Pragmatism*. State Univ. of N.Y. Press, and Cornell Univ. Press, 14 July 1997.

[70] Robin Le Poidevin and Murray MacBeath, editors. *The Philosophy of Time*. Oxford University Press, 1993.

[71] Karl R. Popper. *Conjectures and Refutations. The Growth of Scientific Knowledge*. Routledge and Kegan Paul Ltd. (Basic Books, Inc.), 39 Store Street, WC1E 7DD, London, England (New York, NY, USA), 1963,...,1981.

[72] Arthur Prior. *Changes in Events and Changes in Things*, chapter in [70]. Oxford University Press, 1993.

[73] Arthur N. Prior. *Logic and the Basis of Ethics*. Clarendon Press, Oxford, UK, 1949.

[74] Arthur N. Prior. *Formal Logic*. Clarendon Press, Oxford, UK, 1955.

[75] Arthur N. Prior. *Time and Modality*. Oxford University Press, Oxford, UK, 1957.

[76] Arthur N. Prior. *Past, Present and Future*. Clarendon Press, Oxford, UK, 1967.

[77] Arthur N. Prior. *Papers on Time and Tense*. Clarendon Press, Oxford, UK, 1968.

[78] Wolfgang Reisig. *Understanding Petri Nets Modeling Techniques, Analysis Methods, Case Studies*. Springer, 2013. 230+XXVII pages, 145 illus.

[79] Gerald Rochelle. *Behind time: The incoherence of time and McTaggart's atemporal replacement*. Avebury series in philosophy. Ashgate, Brookfield, Vt., USA, 1998. vii + 221 pages.

[80] David Rydeheard and Rod M. Burstall. *Computational Category Theory*. Prentice-Hall Intl., 1991.

[81] Donald Sannella and Andrzej Tarlecki. *Foundations of Algebraic Semantcs and Formal Software Development*. Monographs in Theoretical Computer Science. Springer, Heidelberg, 2012.

[82] Barry Smith. Mereotopology: A Theory of Parts and Boundaries. *Data and Knowledge Engineering*, 20:287–303, 1996.

[83] Kai Sørlander. *Det Uomgængelige – Filosofiske Deduktioner [The Inevitable – Philosophical Deductions, with a foreword by Georg Henrik von Wright]*. Munksgaard · Rosinante, Copenhagen, Denmark, 1994. 168 pages.

[84] Kai Sørlander. *Under Evighedens Synsvinkel [Under the viewpoint of eternity]*. Munksgaard · Rosinante, Copenhagen, Denmark, 1997. 200 pages.

[85] Kai Sørlander. *Den Endegyldige Sandhed [The Final Truth]*. Rosinante, Copenhagen, Denmark, 2002. 187 pages.

[86] Kai Sørlander. *Indføring i Filosofien [Introduction to The Philosophy]*. Informations Forlag, Copenhagen, Denmark, 2016. 233 pages.

[87] Kai Sørlander. *Den rene fornufts struktur [The Structure of Pure Reason]*. Ellekær, Slagelse, Denmark, 2022. See [88].

[88] Kai Sørlander. *The Structure of Pure Reason*. Publisher to be decided, 2023. This is an English translation of [87] – done by Dines Bjørner in collaboration with the author.

[89] Fei Xiao Tong. *From the Soil — The Foundations of Chinese Society: XiangTu ZhongGuo*. University of California Press, Berkeley 94720, Calif., USA, (1947) 1992. (Fei Hsiao–t'ung: Hsiang t'u Chung–kuo) Translated by Gary G. Hamilton and Wang Zheng. ISBN 0–520–07796–2[3].

[90] Johan van Benthem. *The Logic of Time*, volume 156 of *Synthese Library: Studies in Epistemology, Logic, Methhodology, and Philosophy of Science (Editor: Jaakko Hintika)*. Kluwer Academic Publishers, P.O.Box 17, NL 3300 AA Dordrecht, The Netherlands, second edition, 1983, 1991.

[91] Achille C. Varzi. *On the Boundary between Mereology and Topology*, pages 419–438. Hölder-Pichler-Tempsky, Vienna, 1994.

[92] Carl von Linné. *An Introduction to the Science of Botany*. Lipsiae [Leipzig]: Impensis Godofr. Kiesewetteri, 1748.

[93] Wang Yi. *A Calculus of Real Time Systems*. PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, 1991.

[3] https://en.wikipedia.org/wiki/Fei_Xiaotong

[94] Chao Chen Zhou and Michael R. Hansen. *Duration Calculus: A Formal Approach to Real–time Systems*. Monographs in Theoretical Computer Science. An EATCS Series. Springer–Verlag, 2004.

Part XIV

Appendix

Appendix A

Concepts

Contents

This book touches upon a rather large number of new concepts. In a conventional software engineering setting, and, as in this case, in a technical/scientific book, such an introduction is unusual. I present this chapter because of the large number of new concepts. In order for the reader to find the way around, that reader must be made aware of the background concepts that underlie my treatment of a **new** branch of software engineering, **the domain science and engineering.**

A.1 A General Vocabulary

1. **Abstraction:**

    `Conception, my boy, fundamental brain-work,`
    `is what makes the difference in all art`
    D.G. Rossetti[1]: letter to H. Caine[2]

    Abstraction is a tool, used by the human mind, and to be applied in the process of describing (understanding) complex phenomena.

    Abstraction is the most powerful such tool available to the human intellect.

    Science proceeds by simplifying reality. The first step in simplification is abstraction. Abstraction (in the context of science) means leaving out of account all those empirical data which do not fit the particular, conceptual framework within which science at the moment happens to be working.

[1] Dante Gabrielli Rosetti, 1828–1882, English poet, illustrator, painter and translator

[2] T. Hall Caine, 1853–1931, British novelist, dramatist, short story writer, poet and critic.

Abstraction (in the process of specification) arises from a conscious decision to advocate certain desired objects, situations and processes as being fundamental; by exposing, in a first, or higher, level of description, their similarities and — at that level — ignoring possible differences.

[From the opening paragraphs of [54, C.A.R. Hoare *Notes on Data Structuring*]]

2. **Computer:** A computer is a collection of *hardware* and *software*, that is, is a machine that can be instructed to carry out sequences of arithmetic or logical operations automatically via computer programming [Wikipedia].

3. **Computer Science:** is the study and knowledge of the abstract phenomena that "occur" within computers [DB].

   As such computer science includes *theory of computation*, *automata theory*, *formal language theory*, *algorithmic complexity theory*, *probabilistic computation*, *quantum computation*, *cryptography*, *machine learning and computational biology.*.

4. **Computing Science:** is the study and knowledge of how to construct "those things" that "occur" within computers [DB].

   As such computing science embodies *algorithm and data structure design*, *functional-*, *logic-*, *imperative-* and *parallel programming*; *code testing*, *model checking* and *specification proofs*. Much of this can be pursued using *formal methods*.

5. **Conservative Extension:** An extension of a logical theory is conservative, i.e., conserves, if every theorem expressible in the original theory is also derivable within the original theory [en.wiktionary.org/wiki/conservative_exten-sion].

6. **Divide and Conquer:** In computer science, divide and conquer is an algorithm design paradigm based on multi-branched recursion. A divide-and-conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly [Wikipedia].

   But this book is not about the exciting field of *algorithm design*.

   Yet, the principle of *divide and conquer* is also very strongly at play here: In the top-down analysis of a *domain* into what can be *described* and what is *indescribable*, of *describable entities* into *endurants* and *perdurants*, of *endurants* into *discrete*, *conjoins* and *materials*, of *discrete* into *physical parts*, *structures* and *living species*, and so forth.

7. **Domain Engineering:** is the engineering of *domain descriptions* based on the engineering of *domain analyses* [DB].

8. **Domain Requirements:** are those requirements which can be expressed sôlely in terms of domain concepts.

9. **Engineering:** is the use of scientific principles to design and build machines, structures, and other items, including bridges, tunnels, roads, vehicles, and buildings [Wikipedia].

   The engineer *walks the bridge between science and technology:* analysing man-made devices for their possible scientific properties and constructing technology based on scientific insight.

10. **Epistemology:** is the branch of philosophy concerned with the theory of knowledge – and is the study of the nature of knowledge, justification, and the rationality of belief [Wikipedia].

11. **Formal Method:** By a *formal method* we shall here understand a *method* whose techniques and tools can be understood mathematically.

    For *formal domain, requirements* or *software engineering* methods *formality* means the following:

    - There is a set, one or more, **specification languages** – say for domain descriptions, requirements prescriptions, software specifications, and software coding, i.e., programming languages.[3]
    - These are all to be formal, that is, to have a formal syntax, a formal semantics, and a formal, typically *Mathematical Logic* proof system.
    - Some of the *techniques* and *tools* must be supported by a mathematical understanding.

12. **Hardware:** The physical components of a computer: electronics, mechanics, etc. [Wikipedia].

13. **Intentional Pull:** The concept of intentional pull is a wider notion than that of invariant. Here we are not concerned with pre-/post-conditions on operations. Intentional pull is exerted between two or more phenomena of a domain when their relation can be asserted to **always** hold.

14. **Interface Requirements:** are those requirements which can be expressed in a combination of both domain and machine concepts. They do so because certain entities, whither endurants or perdurants, are **shared** *between the domain and the machine.*

15. **Invariants:** The concept of invariants in the context of computing science is most clearly illustrated in connection with the well-formedness of data structures. Invariants then express properties that must hold, i.e., as a **pre-condition**, before any application of an operation to those data structures and shall hold, i.e. as a **post-condition** after any application of an operation to those data structures.

[3]Most formal specification languages are textual, but graphical languages like `Petri nets [78]`, `Message Sequence Charts [57]`, `Statecharts [50]`, `Live Sequence Charts [51]`, etc., are also formal.

16. **Language:** By *language* we shall, with `[Wikipedia]`, mean a *structured system* of communication. Language, in a broader sense, is the method of communication that involves the use of – particularly human– languages. The 'structured system' that we refer to has come to be known as *Syntax*, *Semantics* and *Pragmatics*.

17. **Linguistics:** By *linguistics* we shall mean the scientific study of language.

18. **Machine:** By a *machine* we shall understand a combination of software and hardware.

19. **Machine Requirements:** are those requirements which can be expressed sôlely in terms of machine concepts.

20. **Mathematics:** By *mathematics* we shall here understand a such human endeavours that makes precise certain facets of language, whether natural or 'constructed' (as for mathematical notation), and out of those endeavours, i.e., mathematical constructions, also called theories, build further abstractions.

21. **Metaphysics:** is the branch of philosophy that examines the fundamental nature of reality, including the relationship between mind and matter, between substance and attribute, and between potentiality and actuality [66] `[Wikipedia]`.

    In this book we stay clear of metaphysics.

22. **Mereology:** is the theory of parthood relations: of the relations of part to whole and the relations of part to part within a whole [39, 82, 91].

    The term 'mereology' is accredited to the Polish mathematician, philosopher and logician Stanisław Leśniewski (1886–1939).

23. **Method:** By a method we shall understand a *set of* **principles** and **procedures** for selecting and applying a *set of* **techniques** using a *set of* **tools** in order to construct an artefact [DB].

24. **Methodology:** is the comparative study and knowledge of methods [DB].

    [The two terms: 'method' and 'methodology' are often confused, including used interchangeably.]

25. **Model:** A mathematical model is a description of a system using mathematical concepts and language. We shall include descriptions[4], prescriptions[5] and specifications[6] using formal languages as presenting models.

[4] as for domains
[5] as for requirements
[6] as for software

26. **Modelling:** Modelling is the act of creating models, which include discrete mathematical structures (sets, Cartesians, lists, maps, etc.), and are logical theories represented as algebras. That is, any given RSL text denotes a set of models, and each model is an algebra, i.e., a set of named values and a set of named operations on these. Modelling is the engineering activity of establishing, analysing and using such structures and theories. Our models are established with the intention that they "model" "something else" other than just being the mathematical structure or theory itself. That "something else" is, in our case, some part of a reality[7], or of a construed such reality, or of requirements to the, or a reality[8], or of actual software[9].

27. **Narration & Formalisation:** To communicate what a domain "is", one must be able to narrate of what it consists. To understand a domain one must give a formal description of that domain. When we put an ampersand, &, between the two terms we mean to say that they form a whole: not one without the other, either way around! In our domain descriptions we enumerate narrative sentences and ascribe this enumeration to formal expressions.

28. **Nondeterminism:** Non-determinism is a fundamental concept in computer science. It appears in various contexts such as automata theory, algorithms and concurrent computation. ...The concept was developed from its inception by Rabin & Scott, Floyd and Dijkstra; as was the interplay between non-determinism and concurrency [Michal Armoni and Mordechai Ben-Ari].

29. **Ontology:** is the branch of metaphysics dealing with the nature of being; a set of concepts and categories in a subject area or domain that shows their properties and the relations between them [36, 37] [Wikipedia].

30. **Operational Abstraction** abstract the way in which we express operations on usually representationally abstracted values. In conventional programming we refer to operational abstract as *procedure abstraction*.

31. **Philosophy:** is the study of general and fundamental questions about existence, knowledge, values, reason, mind, and language. Such questions are often posed as problems to be studied or resolved [Wikipedia].

32. **Pragmatics:** studies the ways in which context contributes to meaning. Pragmatics encompasses speech act theory, conversational implicature, talk in interaction and other approaches to language behavior in philosophy, sociology, linguistics and anthropology [68, 69] [Wikipedia].

33. **Principle:** By a *principle* we shall, loosely, understand *(i) elemental aspect of a craft or discipline, (ii) foundation, (iii) general law of nature, etc* [www.etymonline.com].

[7]— as in domain modelling
[8]— as in requirements modelling
[9]— as in software design

Among basic principles, to be applied across all phases of software development, and hence in all phased of software engineering are those of *abstraction: conservative extension*, *divide and conquer*, *establishing invariants and intentional pull*, *narration & formalisation*, *non-determinism*, *operational abstraction*, *refinement*, and *representational abstraction*. We refer to their definitions in this chapter.

34. **Refinement** is a verifiable transformation of an abstract (i.e., high-level) formal specification into a less abstract, we say more concrete (i.e., low-level) specification or an executable program. Step-wise refinement allows the refinement of a program, from a specification, to be done in stages [www.igi-global.com/dictionary].

35. **Representational Abstraction** abstracts the representation of type values, say in the form of just plain **sorts**, or, when concrete **types**, then in, for example the form of mathematical sets, or maps (i.e., discrete functions, usually from finite definition sets into likewise representationally abstracted ranges), or Cartesians (i.e., groupings of likewise abstracted elements), etc. In conventional programming we refer to representational abstract as *data abstraction*.

36. **Requirements:** By a requirements we understand (cf., [56, IEEE Standard 610.12]): *"A condition or capability needed by a user to solve a problem or achieve an objective"*

    In *software development* the requirements explain what properties the desired software should have, not how these properties might be attained. In our, the *triptych* approach, requirements are to be "derived" from domain descriptions.

37. **Requirements Engineering:** is the engineering of constructing requirements [DB].

38. **Requirements Prescription:** By a requirements prescription we mean a document which outlines the requirements that some software is expected to fulfill.

39. **Requirements Specification:** By a requirements specification we mean the sames as a requirements prescription.

40. **Science:** is a systematic enterprise that builds and organizes knowledge in the form of testable explanations and predictions about the universe [Wikipedia].

    Science is the intellectual and practical activity encompassing the systematic study of the structure and behaviour of the physical and natural world through observation and experiment.

41. **Semantics:** is the linguistic and philosophical study of meaning in language, programming languages, formal logics, and semiotics. It is concerned with the relationship between signifiers — like words, phrases, signs, and symbols — and what they stand for in reality, their denotation [38] [Wikipedia].

    The languages that we shall be concerned with is, on one hand, the language[s] in which we describe domains, here MoLa, and, on the other hand, the language that emerges as the result of our domain analysis & description: a domain specific language.

> There are basically three kinds of semantics, expressed somewhat simplistically:
>
> - **Denotational Semantics** model-theoretically assigns a *meaning*, a *denotation*, to each *phrase structure*, i.e., *syntactic category*.
> - **Axiomatic Semantics** or **Mathematical Logic Proof Systems** is an approach based on mathematical logic for proving the correctness of specifications.
> - **Algebraic Semantics** is a form of axiomatic semantics based on algebraic laws for describing and reasoning about program semantics in a formal manner.

42. **Semiotics:** is the study and knowledge of sign process (semiosis), which is any form of activity, conduct, or any process that involves signs, including the production of meaning [Wikipedia].A sign is anything that communicates a meaning, that is not the sign itself, to the interpreter of the sign. The meaning can be intentional such as a word uttered with a specific meaning, or unintentional, such as a symptom being a sign of a particular medical condition. Signs can communicate through any of the senses, visual, auditory, tactile, olfactory, or gustatory [Wikipedia].

    The study and knowledge of semiotics is often "broken down" into the studies, etc., of *syntax*, *semantics* and *pragmatics*.

43. **Software:** is the is the set of all the documents that have resulted from a completed *software development: domain analysis & description*, *requirements analysis & prescription*, *software: software code*, *software installation manuals*, *software maintenance manuals*, *software users guides*, *development project plans*, *budget*, *etc.*

44. **Software Design:** is the engineering of constructing software [DB].

    Whereas software requirements engineering focus on the logical properties that desired software should attain, software design, besides focusing on achieving these properties *correctly*, also focus on the properties being achieved *efficiently*.

45. **Software Engineering:** to us, is then the combination of domain and requirements engineering with software design [DB].

    This is my characterisation of software engineering. It is at the basis of this book as well as [8–16].

46. **Software Development:** is then the combination of the development of domain description, requirements prescription and software design [DB].

    This is my characterisation of software engineering. It is at the basis of this book as well as [8–16].

47. **Syntax:** is the set of rules, principles, and processes that govern the structure of sentences (sentence structure) in a given language, usually including word order [Wikipedia].

    We assume, as an absolute minimum of knowledge, that the reader of this primer is well aware of the concepts of BNF (*Backus Normal Form*) Grammars and CFGs (*Context Free Grammars*).

48. **Syntax, Semantics and Pragmatics:** With the advent of computing and their attendant programming languages these concepts of semiotics has taken on a somewhat additional meaning. When, in computer & computing science and in software engineering we speak of syntax we mean a quite definite and (mathematically) precise thing. With the advent our ability to mathematically precise describe the semantics of [certain] programming languages, we similarly mean quite definite and (mathematically) precise things. For natural, i.e., human languages, this is not so. s for pragmatics there is this to say. Computers have not pragmatics. Humans have. When, in this book we bring the term 'pragmatics' into play we are referring not to the computer "being pragmatic", but to our pragmatics, as scientists, as engineers.

49. **Taxonomy:** is the practice and science of classification of things or concepts, including the principles that underlie such classification [Wikipedia].

50. **Technique:** By a *technique* we shall, loosely, understand *(i) formal practical details in artistic, etc., expression, (ii) art, skill, craft in work*" [www.etymonline.com]. Classical technique are that of establishing **invariants** and expressing **intentional pull**.

51. **Technology:** is the sum of techniques, skills, methods, and processes used in the production of goods or services or in the accomplishment of objectives, such as scientific investigation [Wikipedia].

    Technology can be the knowledge of techniques, processes, and the like, or it can be embedded in machines to allow for operation without detailed knowledge of their workings. Systems (e.g. machines) applying technology by taking an input, changing it according to the system's use, and then producing an outcome are referred to as technology systems or technological systems [Wikipedia].

52. **Tool:** By a *tool* we shall, loosely, understand *(i) instrument, implement used by a craftsman or laborer, weapon, (ii) that with which one prepares something, etc.* [www.-etymonline.com].

53. **Triptych:** The *triptych [of software development]* centers on the three 'engineerings': domain, requirements and software [DB]. We refer to *The Triptych Dogma* of Page V.

Appendix B

Solutions

Contents

B.0 Introduction

B.1 Domains I

Solution 1 *xxx1:*

B.2 Logic

[We refer to Sect. **2** on page 49.]

Solution 2 *xxx:*

B.3 Sets

We refer to Chapter **3** on page 63.

Solution 3 *XSets:* [We refer to Sect. **3.5.1** on page 70 and Exercise **??** on page ??]

Solution 4 *YSets:* [We refer to Sect. **3.5.1** on page 70 and Exercise **??** on page ??]

Solution 5 *ZSets:* [We refer to Sect. **3.5.1** on page 70 and Exercise **??** on page ??]

B.4 Numbers and Numerals

We refer to Chapter **4** on page 77.

Solution 6 *xxx3:*

B.5 Names and Values

We refer to Chapter **5** on page 89.

Solution 7 *xxx3x:*

B.6 Functions

We refer to Chapter **6** on page 101.

Solution 8 *xxx4:*

B.7 Infinity

We refer to Chapter **7** on page 107.

Solution 9 *xxx5:*

B.8 Cartesians

We refer to Chapter **8** on page 109.

Solution 10 *xxx6:*

B.9 Graphs

We refer to Chapter **9** on page 119.

Solution 11 *Your Home:*

Solution 12 *Your Neighbourhood:*

Solution 13 *National Geography:*

Solution 14 *Properties of Trees:* [We refer to Example 62 on page 131 and Exercise 33 on page 133]

B.10 Lists

We refer to Chapter **10** on page 135.
:

B.11 Maps

We refer to Chapter **11** on page 145.
:

B.12 Types and Sorts

We refer to Chapter **12** on page 153.
:

B.13 Space

We refer to Chapter **13** on page 167.
:

B.14 Geometry

We refer to Chapter **14** on page 173.
:

B.15 Time

We refer to Chapter **15** on page 177.
:

B.16 Structured Expressions

We refer to Chapter **16** on page 185.
:

B.17 Sequentiality

We refer to Chapter **17** on page 189.
:

B.18 Concurrency

We refer to Chapter **18** on page 191.
:

B.19 Applicative Algorithms

We refer to Chapter **19** on page 195.
:

B.20 Imperative Algorithms

We refer to Chapter **20** on page 197.
:

B.21 Concurrent Algorithms

We refer to Chapter **21** on page 199.

:

B.22 Applicative Programming Languages

We refer to Chapter **22** on page 203.

:

B.23 Imperative Programming Languages

We refer to Chapter **23** on page 205.

:

B.24 Concurrent Programming Languages

We refer to Chapter **24** on page 207.

:

B.25 Logic Programming Languages

We refer to Chapter **25** on page 209.

:

B.26 Domain Modeling

We refer to Chapter **26** on page 213.

:

B.27 Petri Modeling

We refer to Chapter **27** on page 215.

:

B.28 Algebraic Modeling

We refer to Chapter **28** on page 217.

:

B.29 Syntax

We refer to Chapter **29** on page 221.
:

B.30 Semantics

We refer to Chapter **30** on page 223.
:

B.31 Pragmatics

We refer to Chapter **31** on page 225.
:

B.32 Domains II

We refer to Chapter **32** on page 229.
:

B.33 Requirements

We refer to Chapter **33** on page 231.
:

B.34 Software

We refer to Chapter **34** on page 233.
:

B.35 Mathematics

We refer to Chapter **35** on page 237.
:

B.36 Natural Sciences

We refer to Chapter **36** on page 239.
:

B.37 Transport

We refer to Chapter **37** on page 241.
:

B.38 Documents

We refer to Chapter **38** on page 245.
:

B.39 Industry

We refer to Chapter **39** on page 249.
:

B.40 Public Services

We refer to Chapter **40** on page 251.
:

B.41 ... Services

We refer to Chapter **41** on page 253.
chapServices:

B.42 Management

We refer to Chapter **42** on page 255.
:

Appendix C

Measures

Contents

C.1 International System of Units

In this section we shall muse about the kind of attributes that are typical of natural parts, but which may also be relevant as attributes of artefacts. Our departure point is that of the *International System of Units, ISU*[1].

Typically, when physicists write computer programs, intended for calculating physics behaviours, they "lump" all of these into the **type Real**, thereby hiding some important physics 'dimensions'. In this section we shall review that which is missing!

[1] `https://en.wikipedia.org/wiki/International_System_of_Units`

The subject of physical dimensions in programming languages is rather decisively treated in David Kennedy's 1996 PhD Thesis [61] — so there really is no point in trying to cast new light on this subject other than to remind the reader of what these physical dimensions are all about.

C.1.1 SI: The International System of Quantities

In physics we operate on values of attributes of manifest, i.e., physical phenomena. The type of some of these attributes are recorded in well known tables, cf. Tables C.1–C.3. Table C.1 shows the base units of physics.

Base quantity	Name	Type
length	meter	m
mass	kilogram	kg
time	second	s
electric current	ampere	A
thermodynamic temperature	kelvin	K
amount of substance	mole	mol
luminous intensity	candela	cd

Table C.1: Base SI Units

Table C.2 on the next page shows the units of physics derived from the base units. Table C.3 shows further units of physics derived from the base units. *velocity* is speed with three dimensional direction and is, for example, given as

- velocity, meter per second with direction: m/s
- acceleration, meter per second squared and *(longitude,latitude,azimuth)* measured in $r$adian: $m/s^2(r,r,r)$

Table C.4 shows standard prefixes for SI units of measure and Tables C.5 show fractions of SI units.

• • •

The point in bringing this material is that when modelling, i.e., describing domains we must be extremely careful in not falling into the trap of modelling physics types, etc., as we do in programming – by simple **Real**s. We claim, without evidence, that many trivial programming mistakes are due to confusions between especially derived SI units, fractions and prefixes.

C.1.2 Units are Indivisible

A volt, kg$\times$m$^2\times$s$^{-3}\times$A^{-1}, see Table C.2, is "indivisible". It is not a composite structure of mass, length, time, and electric current – in some intricate relationship.

• • •

Name	Type	Derived Quantity	Derived Type
radian	rad	angle	m/m
steradian	sr	solid angle	$m^2 \times m^{-2}$
Hertz	Hz	frequency	s^{-1}
newton	N	force, weight	$kg \times m \times s^{-2}$
pascal	Pa	pressure, stress	N/m^2
joule	J	energy, work, heat	$N \times m$
watt	W	power, radiant flux	J/s
coulomb	C	electric charge	$s \times A$
volt	V	electromotive force	W/A $(kg \times m^2 \times s^{-3} \times A^{-1})$
farad	F	capacitance	C/V $(kg^{-1} \times m^{-2} \times s^4 \times A^2)$
ohm	Ω	electrical resistance	V/A $(kg \times m^2 \times s^3 \times A^2)$
siemens	S	electrical conductance	A/V $(kg1 \times m^2 \times s^3 \times A^2)$
weber	Wb	magnetic flux	$V \times s$ $(kg \times m^2 \times s^{-2} \times A^{-1})$
tesla	T	magnetic flux density	Wb/m^2 $(kg \times s^2 \times A^{-1})$
henry	H	inductance	Wb/A $(kg \times m^2 \times s^{-2} \times A^2)$
degree Celsius	^oC	temp. rel. to 273.15 K	K
lumen	lm	luminous flux	$cd \times sr$ (cd)
lux	lx	illuminance	lm/m^2 $(m^2 \times cd)$

Table C.2: Derived SI Units

Physical attributes may ascribe mass and volume to endurants. But they do not reveal the substance, i.e., the material from which the endurant is made. That is done by chemical attributes.

C.1.3 Chemical Elements

The chemical elements are, to us, what makes up MATTER. The *mole*, mol, substance is about chemical molecules. A mole contains exactly $6.02214076 \times 10^{23}$ (the Avogadro number) constituent particles, usually atoms[2], molecules, or ions – of the elements, cf. 'The Periodic Table', `en.wikipedia.orgwiki/Periodic_table`, cf. Fig. C.1. Any specific molecule is then a compound of two or more elements, for example, calciumphosphat: Ca3(PO4)2.

Moles bring substance to endurants. The physics attributes may ascribe weight and volume to endurants, but they do not explain what it is that gives weight, i.e., fills out the volume.

[2]**Mole:** One mole of a substance is equal to 6.022×10^{23} units of that substance (such as atoms, molecules, or ions). The number 6.022×10^{23} is known as **Avogadro's number** or **Avogadro's constant**. The concept of the mole can be used to convert between mass and number of particles.

Name	Explanation	Derived Type
area	square meter	m^2
volume	cubic meter	m^3
speed	meter per second	m/s
wave number	reciprocal meter	m-1
mass density	kilogram per cubic meter	kg/m^3
specific volume	cubic meter per kilogram	m3/kg
current density	ampere per square meter	A/m^2
magnetic field strength	ampere per meter	A/m
substance concentration	mole per cubic meter	mol/m3
luminance	candela per square meter	cd/m^2
mass fraction	kilogram per kilogram	$kg/kg = 1$

Table C.3: Further SI Units

Prefix name		deca	hecto	kilo	mega	giga
Prefix symbol		da	h	k	M	G
Factor	10^0	10^1	10^2	10^3	10^6	10^9
Prefix name		tera	peta	exa	zetta	yotta
Prefix symbol		T	P	E	Z	Y
Factor		10^{12}	10^{15}	10^{18}	10^{21}	10^{24}

Table C.4: Standard Prefixes for SI Units of Measure

C.2 Spatial Measures

C.2.1 Angle

C.2.2 Latitude, Longitude and Altitude

Latitude and Longitude are the units that represent the *coordinates* of a *geographic coordinate system.* **Altitude**, like *elevation,* is the is the *height above* or *depth below sea level*.

Mean sea level[3] (MSL, often shortened to sea level) is an average surface level of one or more among Earth's coastal bodies of water from which heights such as elevation may be measured. The global MSL is a type of vertical datum – a standardized geodetic datum – that is used, for example, as a chart datum in cartography and marine navigation, or, in aviation, as the standard sea level at which atmospheric pressure is measured to calibrate altitude and, consequently, aircraft flight levels. A common and relatively straightforward mean sea-level standard is instead the midpoint between a mean low and mean high tide at a particular location.[4]

[3] https://en.wikipedia.org/wiki/Sea_level

[4] What is "Mean Sea Level"? Archived 21 April 2017 at the *Wayback Machine (Proudman Oceanographic Laboratory)*.

Prefix name		deca	hecto	kilo	mega	giga
Prefix symbol		da	h	k	M	G
Factor	10^0	10^1	10^2	10^3	10^6	10^9
Prefix name		tera	peta	exa	zetta	yotta
Prefix symbol		T	P	E	Z	Y
Factor		10^{12}	10^{15}	10^{18}	10^{21}	10^{24}
Prefix name		deci	centi	milli	micro	nano
Prefix symbol		d	c	m	μ	n
Factor	10^0	10^{-1}	10^{-2}	10^{-3}	10^{-6}	10^{-9}
Prefix name		pico	femto	atto	zepto	yocto
Prefix symbol		p	f	a	z	y
Factor		10^{-12}	10^{-15}	10^{-18}	10^{-21}	10^{-24}

Table C.5: SI Units of Measure and Fractions

C.3 How to Measure

Metrology[5] is *the science of measurement, embracing both experimental and theoretical determinations at any level of uncertainty in any field of science and technology, as defined by the International Bureau of Weights and Measures* (BIPM, 2004,).

much more to come

C.4 Precision, Probability, Statistics and Fuzziness

to be written

C.4.1 Precision

to be written

C.4.2 Probability

to be written

C.4.3 Statistics

to be written

C.4.4 Fuzziness

to be written

[5] https://www.nist.gov/metrology

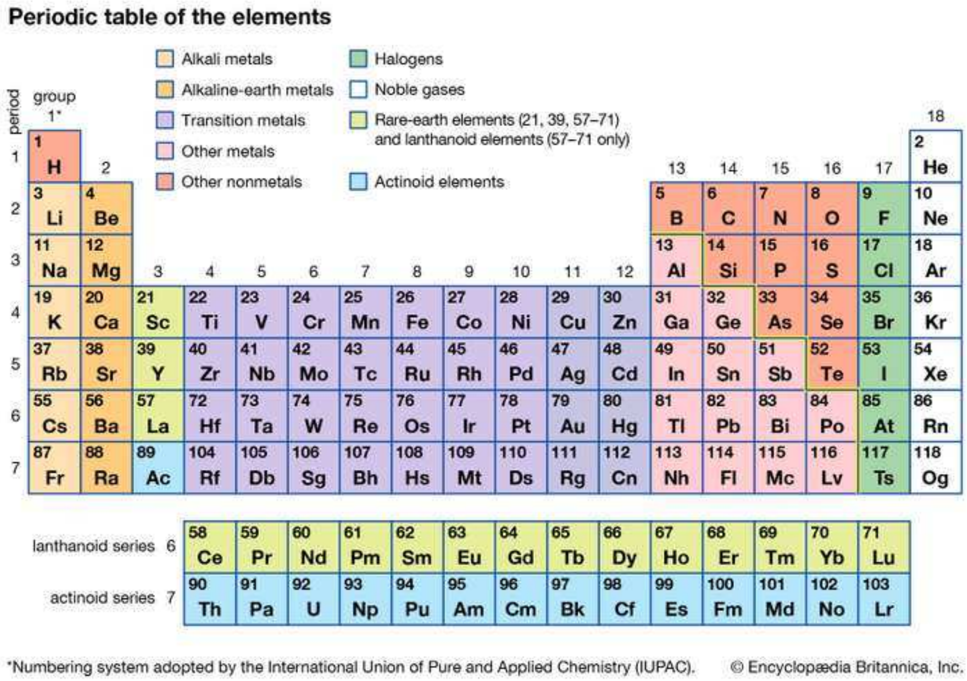


Figure C.1: Periodic Table

C.5 Summary & Conclusion

to be written

Appendix D

MoLa: The Modeling Language

Contents

D.1 Identifiers

to be written

D.2 Constants

Constants name values that do not change,

54. The Boolean **true** and **false** names constants. See Item 83 on page 311.
55. The characters **a, ..., z, A, ..., Z** are constants.
56. Any text is a constant: **"any_text_is_a_constant**.
57. The empty set {} is names a constant. See Item 89 on page 312.
58. Numerals are constants. See Item 68 on page 310.
59. The empty list <> names a constant. See Item 97 on page 313.
60. The empty map [] names a constant. See Item 98 on page 313.
61. Any expression, a set, Cartesian,list or map enumeration composed solely from constants is a constant.

54. Const ::= **true** || **false**
54. || **Char**
54. || **Text**
57. || Numeral
58. || {}
59. || $\langle\rangle$
60. || [ ]
61. || [see explanation next]

We comment on Item 61: in Sects. **D.7.2** on page 312, **D.7.4** on page 313, **D.7.5** on page 313, and **D.7.6** on page 313, we formally outline the syntax (and, informally, the meaning) of sets, Cartesians, lists and maps. Wherever, in these expression syntaxes, the terms stand for constants, the corresponding expression is a constant!

D.3 Names & Identifiers

Names denote types or values of elements of a type (see first line of Sect. **D.6** below) or other[1]. We give names to Booleans, numbers (i.e., numerals), sets, Cartesians, lists, maps and functions. We have not yet given a formalized introduction to *characters* and *texts*. So we do that next!

We use to term **name** to express that we name something, i.e., that the name designates, denotes, that "thing"; that is, the **semantics** of the name is that thing!

[1] By 'other' we mean: of *states*, *assignable variables*, or *channels*.

And we use the term **identifier** to express a **syntactic** quantity, usually a sequence of characters.

Which characters we then choose to allow as elements of an identifiers character sequence is then a matter of **pragmatics**.

Henceforth we shall present the concrete syntax of identifiers, whether of types or of values or other[2].

62. 'a', 'b', ..., 'z' are alphabetic characters, AlphaChar.

63. '0', '1', ..., '9' are digits, Digit.

64. An alphabetic character or a digit is a character.

65. The alphabetic characters are identifiers, ID.

66. An alphabetic character followed by a

    (a) [possibly '_' prefixed underscore and then]

    (b) by a suffix identifier is an identifier.

67. Any non-empty sequence of characters

    (a) and possibly properly in-fixed underscores, '_',

    (b) is a suffix identifier, SuffixID.

```
62.  AlphaChar ::= a  || b  || ...  || z
63.  Digit ::=  0  || 1  || ...  || 9
65.  Character ::= AlphaChar  || Digit
65.  ID ::= AlphaChar
66b.      || AlphaChar  SuffixID
66a.      || AlphaChar  _  SuffixID
67.  SuffixIdD ::= Character
67a.       || Character  _  SuffixID
67b.       || Character  SuffixID
```

D.4 Patterns

to be written

[2] By 'other' we mean: of *states*, *assignable variables*, or *channels*.

D.5 Numerals

Digits are defined in Item 63 above.

68. Digits are numerals.
69. Any digit sequence is a numeral.
70. Any digit sequences separated by a *period:* '.' is a numeral.
71. A digit is a digit sequence.
72. A digit followed by a digit sequence is a digit sequence.

68. Numeral ::= Digit
68. ‖ DigitSeq
70. ‖ DigitSeq . DigitSeq
71. DigitSeq ::= Digit
72. ‖ Digit DigitSeq

D.6 Types

We shall consider types to be a special kind of sets (where we do not consider what is so special about theses sets). The symbol '::=' denotes definition, and the symbol '‖' denotes syntactic alternative.

73. Type_Expr ::= **Bool** ‖ **Nat** ‖ **Int** ‖ **Real** ‖ **Char** ‖ **Text**
74. ‖ A-**set**
74. ‖ A-**infset**
75. ‖ A$\times$B$\times$...$\times$C
76. ‖ A^*
76. ‖ A^ω
77. ‖ A $\xrightarrow[m]{}$ B
78. ‖ A $\rightarrow$ B
78. ‖ A $\xrightarrow{\sim}$ B
79. ‖ Type_Expr_1 | Type_Expr_2 | ... | Type_Expr_n
80. ‖ ID [3]
81. Type_Definition ::= ID = Type_Expr
82. ‖ ID :: (sel_a:A sel_b:B ... sel_c:C)

Type expressions expresses types. Type definitions define types by giving names, TId, to types.

[3] For ID see Item 65 on the previous page

73. A type expression expresses either that of a Boolean, or a natural number, or an integer, or a real, or a characters, or a text;
74. or a type expression expresses finite or also infinite sets;
75. or a type expression expresses Cartesians of types A, B, ..., C;
76. or a type expression expresses finite or infinite lists of element of type A;
77. or a type expression expresses maps from elements of type A to elements of type B;
78. or a type expression expresses total, respectively partial functions from elements of type A to elements of type B;
79. or a type expression expresses the union type of elements of types Type_Expr_1, or types Type_Expr_2, ..., or Type_Expr_n;
80. or a type expression is a type name expressing whatever the definition of that type name expresses.

A type definition consists of three elements: a type name, ID, and type definition symbol, either '=' or '::', and a right hand side.

81. The right hand side is a type expression. The type being defined, by some name in ID, is that of the type expression.
82. The right hand side is a special Cartesian type expression. By (sel_a:A sel_b:B ... sel_c:C) we mean a Cartesian (omitting the infix $\times$) whose individual elements can be **sel**_eted by the selectors, i.e., the [primitive] functions prefixing the Cartesian component type identifiers (sel_A, sel_B, ..., sel_C).

D.7 Expressions

D.7.1 Boolean

Boolean expressions are such expressions which evaluate to a Boolean value.

83. **true** and **false** are Boolean expressions.
84. A triplet of two Boolean expressions with an infix Boolean connective is a Boolean expression.
85. The existential, unique existential and universal predicate over variables a, b, ..., c of appropriate types A, B, ..., C, are Boolean expressions.
86. These are the Boolean connectives: $\sim$, $\wedge$, $\vee$, $\supset$.

87. A triplet of a pair of set, or a pair of Cartesian, or a pair of list, or a pair of map expressions in-fixed by a set, Cartesian, list or map relational is a Boolean expression.

88. $=, \neq, \subset, \subseteq$ are set relational operators.

89. $=, \neq$ are Cartesian relational operators.

90. $=, \neq$ are list relational operators.

91. $=, \neq$ are map relational operators.

83. Bool_Expr ::= **true** || **false**
84. || Bool_Expr × Bool_Conn × Bool_Expr
85. || ∃ a:A,b:B,...,c:C • Bool_Expr
85. || ∃! a:A,b:B,...,c:C • Bool_Expr
85. || ∀ a:A,b:B,...,c:C • Bool_Expr
87. || Set_Expr Set_Rel Set_Expr
87. || Cart_Expr Car_Rel Cart_Expr
87. || List_Expr List_Rel List_Expr
87. || Map_Expr Map_Rel Map_Expr
86. Bool_Conn ::= ∼
86. || ∧
86. || ∨
86. || ⊃
88. Set_Rel ::= =
88. || ≠
88. || ⊂
88. || ⊆
89. Cart_Rel ::= =
89. || ≠
90. List_Rel ::= =
90. || ≠
91. Map_Rel ::= =
91. || ≠

D.7.2 Sets

Set expressions are such expressions which evaluate to a set value.

92. The empty set is a set expression.

93. Set enumeration is a set expression.

94. Set comprehension is a set expression.

95. A set expression followed by a set [producing] operator followed by a set expression is a set expression.

96. The operators $\cup, \cap$, $\backslash$ and / are set producing operators.

92. Set_Expr ::= {}
93. || { Expr , Expr , ... , Expr }
94. || { Expr[4] | ID:Type_Expr,...,ID:Type_Expr : Bool_expr[5]}
95. || Set_Expr Set_Op Set_Expr
96. Set_Op ::= $\cup$ || $\cap$ || \ || /

D.7.3 Arithmetics

Arithmetic expressions are such expressions which evaluate to a number value.

D.7.4 Cartesians

Cartesian expressions are such expressions which evaluate to a Cartesian value.

D.7.5 Lists

List expressions are such expressions which evaluate to a list value.

97. The empty list is a list expression.

D.7.6 Maps

Map expressions are such expressions which evaluate to a map value.

98. The empty map is a map expression.

[4]This expression is expected to contain the [free] variables defined, hence bound to, the values of the sequence of one or more type expressions immediately to the right of the BAR.

[5]This Boolean expression is expected to contain the variables bound to the values of the sequence of one or more type expressions between the BAR and the Boolean expression.

D.8 Functions

D.9 Structured Expressions

D.10 Imperative Statements

D.11 CSP: Communicating Sequential Processes

Appendix E

Indexes

E.1 Definitions

E.2 Concepts

E.3 Domain Examples

E.4 Other Examples

E.5 Symbols

E.6 Text Changes

Appendix F

Log Book

- 26.6.2023: Looking back:
  - Mid June 2023: Revived the idea that led to this document
  - 22.6.2023: Having completed yet a swing through the editing of [27] I established the `main, logic-I, numbers-I, functions-I, types-I, calculations-I, sets`, `Cartesians`, `lists` and `maps` files.
  - 23.6.2023:
  - 24.6.2023: Established the `chapAxiomSystems, chapClosing, chapFormalBases, chapSyntax, chapSemantics, chapPragmatics, chapFormalBases, chapVerification, chapRecursiveFunctionTheory` files. Established the part structuring.
  - 25.6.2023: Established the `chapRoadTransport, chapSpace, chapTime, chapAlgebra, chapRoadTransport, chapSpace, chapMathematicalLogic`

  Established the `chapLogBook` file.
  Renamed a part and some chapters to reflect the change of focus: introducing the *Triptych* dogma as a leading principle of this document.
  Established the `partTriptych, chapRequirements, chapSoftware` files.
  Wrote a bit in Chapters **0** and **2**.

- 27.6.2023: Took the full consequence of the emphasis on domains by establishing appendix chapters `chapRiversCanals, chapStockExchange, chapCreditCard, chapFinanceIndust`
  Perhaps more to come.

- 28.6.2023: Added appendix chapter C, Intl.Sys.of Units plus text from earlier writings – yet to be edited.
  Added sections to Chapter 1: Didactics, Method & Methodology, Computer & Computing Science.

- 29.6.2023: Replaced contemplated separate chapters on *Functions*, resp., *Types* by one, "smaller" Chapter **6** on *Functions and Types – An Introduction*.

- 30.6.2023: Text for Sect. **6** on page 101.
  Revision/simplification to Chapter **4**.
- 1.7.2023: Edited new Appendix Chapter A *Concepts*. Taken from [23].
- 2.7.2023: Further edit on Appendix Chapter A, Vocabulary.
  Work on Chapter **3**, Sets. First "completion".
- 3.7.2023: Moved chapter on sets up just before the chapter on numbers.
  Inserted definitions of the *fac*torial and the *fib*onacci functions into Chapter **6**.
  Work on Chapter **8**, Cartesians. First "completion".
- 4.7.2023: Work on Chapter **10**.[1]
- 5.7.2023: Work on both Power Plant document and this: Inserted new chapters: Infinity **7** and Graphs and Trees **9**.
- 6.7.2023: Work on Power Plant and this doc.: sets, graphs, ...

  July: Filling in various chapters
- 29.7.2023: Putting a new chapter, Chapter **1**, as the first, real chapter of the book. Beginning to consistently begin "all" chapters with **Motivation** text.
- 30.7.2023: MoLa. **To be done**, sooner or later: streamline Chapters **2**–**4** and **8**–**11** presentation of logic, number, set, Cartesian, list and map presentations.
- 6.8.23: Plans for several new chapters:

```
Physics with subsections on                     ...
     Mechanics
     Electricity                            Geography
     Thermodynamics                             Geodecy
     ...
                                            Geology
Botanics, Zoology, Biology                       Minerals
     Botanics                                    Liquids
          Herbaria
          Plantebestemmelse                  Meteorology
          Formering ...                           Weather
      Zoology                                         Statically
           ...                                        Dynamocally
      Biology                                       Renewal
           Conception                                    ...
           Inherited attributes     Etcetera
           Family "trees"
```

[1]Started studying and sketching a domain model for Nuclear Power Plants!

And a chapter on Documents:

```
Two facts:                            ****

  1. There are the physical, paper    Employments certificates
      or electronic "official", i.e., Job references
      concrete documents              Education certificates
      listed below, and               Honourary certificates: orders
                                      Travel: bus, train, air, ship tickets
  2. There is the abstraction of      Savings certificates
      these, say in the form of a     AyPurchase certificates
      Relational  database            Hotel and restaurant b
                                      Utility contracts:
Now: specify a query system               Water, gas, electricity, radio/TV
for searches/interrogation                licence, naintenace, service
                                          contracts, staff contracts
```

Also somewhere in a chapter:

```
Bookkeeping                           Projects:
     Accounts                              Plans
     Accounting                            Planning
     Profit and Loss                       Monitoring
                                           Resource-allocation
Resource-allocation                        [Bookkeeping]
```

- 08.08.23: Introduced text **Change** footnotes, also to [2] Index, cf. Sect. **E.6** on page 323 with possibility of using change start and end margin delimiters [and].

  Worked on Sect. **0.13**.

- 14.08.2023: Added new chapter, Chapter **43**, on *Ontology & Taxonomies*
  Decided to attempt three "CACM" papers:

  – Domain Science: Modeling – based on Chapter **1**,
  – Domain Science: Facets – based on similarly named chapter of [23], and
  – Domain Science: Ontology & Taxonomies – the "parallel" to Chapter **43**.

- 17.8.2023: "Completed" a first writing of the 'Perdurants' section.
  Sort out overlap sections in chapter on Types and Sorts: Type Expressions ...

[2] **Change:** *New text inserted: LogBook*