Theory of Meaningful Information

Boris Sunik

boris.sunik@generalinformationtheory.com

The Theory of Meaningful Information (abbr. ***TMI***) is a general theory that can be applied to information of every type, level and complexity. TMI explains the nature and functionality of information and enables the production of relevant definitions regarding language and knowledge, which remain operative also in the case of non-human languages and knowledge systems. Every piece of information, expressed in any natural language or whatever formal system, can also be expressed in the terms of TMI's concepts without losing any of the semantics. TMI contains the axiomatic theory of intelligence permitting the explanation and reproduction of intelligent algorithms of both natural and artificial intelligences and the theory of the universal representation language, which actually allows such a language to be constructed.

Preface

This theory focuses on meaningful information and intelligence, following a rather unconventional way of problem solving. Hence, a few words regarding its approach even before starting the book. In its most general form, this approach can be formulated as follows:

- The only kind of meaningful information recognized by human beings is a voiced or written text composed in one of the human languages. While the natural languages (English, German, Chinese, etc.) possess unrestricted expression power and unlimitedly extensible vocabulary, they produce ambiguous texts that cannot be adequately understood by a computer.
- Unambiguous human languages can be found among artificial languages. All these languages have limited expression abilities, and only one kind of them, namely the high-level programming languages, are actually used. The expression power of such a language is restricted to a representation of binary sequences (bits, bytes and sequences of bytes) and operations with them. Thus, the notion "car" can routinely be used in English to designate any real or imaginary car, but the definition of such a notion in a C++ program, actually designates a sequence of bytes residing in the storage of a computer running this program.
- C++ is the most prominent programming tool of our time. It is the only compiling language actually used and most of interpreting languages such as C#, Java, JavaScript and so on also follow its syntax and conceptual system. Most complex applications, including AI algorithms, are programmed in this language. However, similar to other programming languages, C++ cannot be used for specification of the contextual information of programmed algorithm, like rules of program use, characteristics of associated external objects existing outside of computer storage and so on.
- TMI defines the universal representation language ***T**,* which can be viewed as an extension of C++ by non-executable representations. In contrast to C++, ***T*** has an unrestricted representation world and allows the explicit specification of arbitrary real or imaginary entities in terms of traditional C++ concepts as class, object, function, parameter etc. This language is able to represent both executable code and non-executable specification of external entities.

Thus, consider a modern robot with a lot of processors controlling its various parts. Currently, the code base characterizing such a robot consists of a bunch of software programming these processors, but in the case of ***T*** it can be a single solution specifying both the executable software and non-executable specifications. The latter part can describe the rules according to which this robot interacts with different external objects, the structure and functionality of these objects, procedures of information collection and so on.

The TMI view of intelligence is based on conceptualization of information-related features of bio-organisms. The ability for information utilization is already present in primitive multicellular organisms and its steady evolutionary development led to the creation of its most complex form – intelligence. While amphibians and reptiles rarely possess any intelligence, this feature exhibits itself in mammals, reaching its peak with Homo sapiens. TMI defines a model called Homo Informaticus, which can be used for the formal representation of the complete intellectual and physical activities of a human being.

TMI allows fixing the extremely dangerous drawbacks of modern neural networks resulting from their ability to produce new concepts. As known, the practical application of neural networks became possible when researchers introduced networks with many neuron layers which made pattern recognition really effective. The specific of this technic is that new knowledge can be obtained from information already processed by a neural network. It just has to analyze the obtained values once more and get new patterns, after which the process can be repeated again. Taking into account the complexity and persistent sophistication of AI systems, this feature unavoidably leads to a situation in which a neural network can break out of control, for example, because it will analyze its own behavior, and change it due to this or some other reason.

TMI offers the ultimate fix to this problem by allowing specification of the complete application environment of a neural network. The code, containing the formal definition of all concepts and instances associated with the network's task, can be used by a supervising controller, which will disable any network's attempt to generate a concept not included in the fixed list of allowed entities.

Table of Contents

1. Introduction

We live in the age of applied sciences. Consider the role of applied biology responsible for the unbelievable progress in medicine or applied physics and applied chemistry, forming the basis of modern industry and modern infrastructure in general. It is nearly impossible to imagine our society without the applications of social sciences like economics, sociology, social psychology, political science, criminology and so on. Frequently ignored, however, is the fact that the boom of applied sciences was only possible because of the fundamental sciences, without which the development of the last 150 years would never have occurred.

The only major type of activities remaining without adequate theoretical conceptualization is that which is concerned with meaningful information. While technical aspects of information transmission are effectively covered by the information theory of Claude Shannon, the meaningful information never possessed any relevant theoretical support. Despite more than a hundred years of research devoted to the formalization of meaning, no research dedicated to the subject was ever able to produce an applicable theory having any impact on the practical information-related activities.

This situation is the result of a deep chasm between theoretical research and practical application over the last few decades. These fields have mostly developed independently from one another. In order to take the necessary step forward in understanding and developing a theory of meaningful information, there is the need to reunite them.

In the theoretical sphere, the first works devoted to this subject had already started in the 19th century with the definition of semantics. In the next century, they became very extensive and were extended to the heterogeneous field including multiple ideas from computing, cognitive sciences, linguistics, medicine and others. Regretfully, a vast amount of energy was used to build a bridge to nowhere - the situation also remains the same in the present century, in which software developers obviously seems to have lost interest in this research because of its complete uselessness.

The failure of theory however never hindered applied developments, which happened to be really effective. The only actual working methods and tools ever employed for representation and processing of meaningful information are the mainstream programming languages used for the creation of both conventional and intelligent applications. Currently their number barely exceeds a dozen, with one of them – C++ - playing an especially outstanding role.

This language, also designated as C/C++, is the only compiled language widely used at this time. Other mainstream programming languages are interpreters, most of them also implementing certain variants of the C++ conceptual system (C# and Java are the most prominent examples of this approach). All mainstream languages were developed in a purely pragmatic way without applying whatever scientific methods[1] and are acquired by programmers intuitively like driving a car. Learning by doing is highly effective in practical work, however, the concepts formed in the heads of specialists in this inductive way are based on tacit knowledge and do not permit deep scientific conceptualization.

The discrepancy between academic research and the realities of everyday software development results from the fundamentally incompatible approaches. While the academic research never succeeded in creating the effective programming tools based on the mathematically correct solutions, the practical programming occurred to be extremely successful by employing programming languages built on non-mathematical basis.

It would be incorrect to say that practical programmers were never interested in scientific research, in reality though, this disinterest is a rather recent phenomenon developed over the last thirty years. A completely different spirit prevailed at the start of the computer era when programmers struggled to find an applicable alternative to a very exhausting and slow machine code programming. Science was seen as the only hope and after the first high-level programming language FORTRAN appeared, the practitioners became theoreticians and produced a rigorous mathematized theory of the programming language. This resulted in viewing all non-mathematical approaches as supposedly non-scientific.

Such an interpretation of programming was not accidental, as the first computers were developed for mathematical calculations and, as a result, most programmers of that time were mathematicians and logicians who viewed programming as a branch of mathematics. The obvious divergence between scientific concepts and the realities of everyday software development was not perceived as a major problem, because mainstream programming languages were considered a temporary solution needed until the theoretically correct programming tools reached maturity. Neglecting mainstream programming went so far that theoreticians failed to produce a generally accepted definition of a programming language because such a definition could not be made in the conceptual coordinates of recognized theories.

This not really effective approach was tolerated in the times of expensive mainframes, but completely lost its appeal after the appearance of C++ and cheap personal computers. In retrospect is clear that a zero output from many decades of intensive mathematic-driven research is the ultimate proof of the wrongness of its key postulate once (presumably) formulated by Galileo Galilei: "The Book of Nature is written in the language of mathematics". This idea widely present in distinct disciplines actually killed the

[1] A single exclusion of this rule is SQL, which is however, a specialized language whose representation domain can be nicely described with mathematical means. See more in 5.4.1.

theoretical programming by automatically rejecting all non-mathematical concepts as unscientific. While no self-respecting scientist will ever study unscientific material, official science is basically unable to leave its convenient blind alley and propose any adequate concepts. While no self-respecting scientist would ever engage in non-scientific studies, official science has basically turned a blind eye to the reality of programming and continue to do so today.

The success of pragmatically developed programming concepts, however, does not mean that proper fundamental software research is no longer considered. On the contrary, it has probably never been as topical as it is now, due to the immense damaging potentiality possessed by failed and seemingly long forgotten theories of the traditional computer science. The reason is the AI theory, which was originally developed as a branch of the latter, reuses many of its results, and is likely obsessed with mathematically correct methods. This theory has finally arrived now - thousands of articles describing miscellaneous formal concepts are printed in multiple AI publications every year (2-3 examples in footnote). That said, however, does not bring us one iota closer to the understanding of what AI is and how it functions.

The results of this dead-end campaign is already seen in the level of discussions about the dangers of AI development. Not just average AI developers, but also such world-known prominent figures as multibillionaire Elon Musk, the late physicist Stephen Hawking and computer scientist Ray Kurzweil, have weighed in on the subject. However, they are also absolutely helpless and their opinions completely lack any theoretically sound arguments.

Currently, AI specialists operate with commonsense considerations and imprecise historical examples[2] and if this situation is allowed to continue, there is a pretty good chance that in the not so distant future, a team of AI developers might underestimate the self-learning capabilities of their own creation and bring an end to the era of human civilization on this planet.

This book attempts to overcome the problem by proposing the Theory of Meaningful Information (abbr. TMI), which is created by extending the universal methods of information formalization developed in programming outside of its original domain. TMI is a general information theory which includes:

- The information theory providing the OO view of real and imaginable worlds;
- The intelligence theory that defines the features of both natural and artificial intelligences and allows the formal specification of the strong AI;

[2] See more in Section 4.1.2.

- The language theory, which includes the theory of the universal representation language allowing the universal representation language T to be formulated.
- The first theory is a general conceptual system, the two following theories are its subsets devoted to the features of intelligence and languages respectively.

TMI is an axiomatic theory, defining the key notions *information*, *knowledge*, *intelligence*, *language*, *sign* solely on the basis of their functional characteristics. In this, it differs from the other approaches of information science restricting these terms to the entities of their interest, as do most of the information-related theories understanding a language as a natural human language, knowledge as human knowledge etc.

The theory is based on an overall definition of information, which can be applied to information of every kind, level and complexity. The definition of information enables a view of the world in terms of objects, actions, relations and properties. Information is considered as the feature manifesting itself in the relations between certain real world entities. TMI provides both the uniform view of all kinds of information and the universal language actually supporting the requested representations.

TMI does not employ already known conceptions and approaches devoted to this subject and basically rejects all mathematically based concepts of meaningful information as irrelevant. This theory cannot be assessed in terms of approaches originating from the domain of formal logic and mathematics as first order calculus, semantic nets, conceptual graphs, frames and ontologies, which constitute the basics of formal languages traditionally used in the realm of knowledge representation like OWL (Smith, Welty, and McGuiness 2004), Cycl (Parmar 2001), KM (Clark and Porter 1999), DART (Evans and Gazdar 1996) and others.

Chapter 2 contains the problem statement.

Chapters 3 - 5 detail theories of information, intelligence and language respectively. Chapter 6 introduces the language T.

Examples of representations are presented in Chapter 7.

2. Problem Statement

2.1. The Classical Information Sciences

The word "information" is derived from the Latin word "informatio" meaning to form, to put in shape, which designated physical forming as well as the forming of the mind, i.e. education and the accumulation of knowledge. This relatively broad meaning was narrowed down in the middle Ages when information equated into education. It had been continually refined up until the first half of the twentieth century when it was equated with "transmitted message".

Today, various kinds of information have been intensively studied by numerous information-related sciences like communication media, information management, computing, cognitive sciences, physics, electrical engineering, linguistics, psychology, sociology, epistemology and medicine. While every discipline concentrates on specific features of information and ignores others, they produce multiple incompatible approaches making generalizations of the studied subject virtually impossible. This is even more complicated by the widespread practice of defining information with the help of other concepts like knowledge and meaning, while simultaneously ignoring related concepts like data, signs, signals.

Taking into account that the absolute majority of informational studies are trivial research without any pretension for broad generalizations, it is no wonder that the Information Age — as often referred to as nowadays — has failed until now to produce a general information theory despite several ambitious attempts in developing universal approaches.

2.1.1. Information Theory of Claude Shannon

The theory was very efficient in solving practical problems of electrical engineering. Though the designation "Information Theory", under which Shannon's work is commonly known, is mainly a misnomer spontaneously produced by the scientific community on the surge of popularity of his approach. Shannon himself never referred to his work in such a manner. The name Shannon gave his theory in the first draft in 1940 was "Mathematical theory of communication" and he retained the same name in a publication in 1963 (Claude Elwood Shannon and Weaver 1963).

He also deliberately evaded consideration of the semantic characteristics of information stressing the aforementioned publication: "Frequently the messages have meaning; that is they refer to or are correlated according to some system with certain physical or conceptual entities. These semantic aspects of communication are irrelevant to the engineering problem" (Claude Elwood Shannon and Weaver 1963). The citation of Weaver from the same publication "The word information, in this theory, is used in the special sense that must not be confused with its ordinary usage. In particular, information must not be confused with meaning...".

These citations actually define the limits of Shannon's theory concerned with the processes of information transmission. This theory however showed no interest in studying the main characteristic of information, which is the ability to designate meaning.

In some paradoxical ways, the channel model of Claude Shannon (**Error! Reference source not found.**, author's own illustration) is very precise in defining the semantics' components of information — they are exactly those parts of the complete picture, which are omitted in this model. That is to say, the sender's semantics ends before (left of) "Message Source" and the receiver's semantics starts only after (right of) "Destination".

Shannon's channel model.

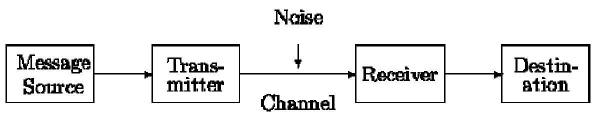

Figure 2-1

Furthermore, the subject of Shannon's Theory cannot even be considered as the attribute of information because primitive information entities can be generated and consumed without any transmission processes whatsoever when the message source and the message destinations are the same (see 3.1.7 for details).

In fact, Shannon's theory is no more than a specialized theoretical framework restricted to the task of transmitting information from one place to another.

2.1.2. Semantic Studies

Another approach is represented by semantics, which studies the meaning of signs and relations between different linguistic units as homonymy, synonymy, antonyms, polysemy and so on. A key concern is how meaning attaches to larger parts of text, possibly as a result of the composition from smaller units of meaning. Traditionally, semantics have included the study of sense and denotative reference, truth conditions, argument structure, thematic roles, discourse analysis, and the linkage of all of these to syntax.

The study of semantics started in the 19th century. Among the early pioneers in this field were Chr. Reisig, who founded the related field of study "semasiology" concerned with the meanings of words, and the French philologist Michel Jules Alfred Bréal, who proposed a "science of significations" that would investigate how sense is attached to expressions and other signs. (Lyons 1977, 619).

In 1910 the British philosophers Alfred North Whitehead and Bertrand Russell published the *Principia Mathematica*, which strongly influenced the Vienna Circle, a group of philosophers who developed the rigorous philosophical approach known as logical positivism. Along with the German mathematician Gottlob Frege, Bertrand Russell also expanded the study of semantics from the mathematical realm into the realm of natural languages. Logical Positivism, the contemporary current of thought trying to create criteria to evaluate statements as true, false or meaningless, thereby making philosophy more rigorous, was concerned with the ideal language and its characteristics, whereas natural languages were regarded as more primitive and inaccurate.

In contrast, the philosophy of "ordinary language" (Ludwig Wittgenstein) saw natural language as the basic and unavoidable matrix of all thought, including philosophical reflections. In his view, an "ideal" language could only be a derivative of a natural language.(Lyons 1977, 140).

Modern approaches initiated by theories of the U.S. linguists Zellig S. Harris and Noam Chomsky went to the development of generative grammars, which provided a deeper insight into the syntax of the natural languages by demonstrating how sentences are built up incrementally from some basic ingredients.(Yule 2006, 101)

The fundamental restriction of semantics making it unfit as the base of any universal information theory is its exclusive connection to a specific kind of meaning which is the one transmitted with the help of a natural language by human beings. Despite the fact that this is the more complex kind of meaning it is surely not the only possible. Types of meaning basically ignored by semantics are tacit knowledge possessed by human beings, non-human knowledge acquired and used by various multicellular and even some unicellular organisms as well as knowledge of computers.

Another restriction of semantics is that it does not take into account the variability of meaning. Transmission of meaning with the help of some language signs represents only one meaning's aspect. Another is the creation and the extension of abilities for expressing and understanding meaning. A newborn baby does not possess knowledge and can neither send nor receive meaning with the help of a natural language; however its ability to send and receive some meaning will evolve during his/her lifetime. By ignoring the variability of meaning, semantics deprives itself from any chance in understanding its nature.

2.1.3. KR Methods

The third approach is embodied by knowledge representation, which is an area of artificial intelligence concerned with the methods of formal representation and use of knowledge. According to (Sowa 2000) knowledge representation is the application of logic and ontology to the task of constructing computable models for some domain. It is a multidisciplinary subject that applies theories and techniques from: a) logic defining the formal structure and rules of inference; b) ontology defining the kinds of things that exist in the application domain; and c) computation that supports the application. Logic encompasses all methods and languages of formal data representations like rules, frames, semantic networks, object-oriented languages, Prolog, Java, SQL, Petri nets, Knowledge Interchange Format (KIF)(Genesereth 1998), Knowledge Machine (KM), Cycl and so on.

The methods of KR partially intersect with those of semantics and its goals are essentially the same, focusing on the formal representation of human knowledge while ignoring knowledge of other kinds.

2.1.4. Miscellaneous Approaches

The last group of approaches consists of miscellaneous works targeting general information theory although without any significant success

Numerous attempts at producing a feasible formalization and generalization of meaningful information are based on Shannon's theory, e.g. (Lu 1999), (Losee 1997).

Generalizations of existing mathematical approaches consolidated into one general theory based on probability was proposed by Cooman (Cooman, Ruan, and Kerre 1995). Keith Devlin (Devlin 1995) tried to build a mathematical theory of information in the form of "a mathematical model of information flow".

Tom Stonier (Stonier 1990) proposed a theory, in which he considered information to be a basic property of the universe much like matter and energy.

At the other end of the scale, there are approaches denying the mathematical core of information and describing the world as consisting of entities and interactions, a reflected relationship of which is information (see (Markov, Ivanova, and Mitov 2007)). Many works targeting the theme are not precise enough and often produce a vague conception of the subject (Burgin 1997).

Building a unified information theory is considered by (Flückiger 1997) and (Fleissner and Hofkirchner 1996).

In summary, the theories developed in Semantics, Knowledge Representation as well as the Shannon's Information Theory are specialized concepts designated for the representation of certain kinds of information or certain stages of information processing. The group of theories (point 4) targeting the creation of a united information theory has never been able to develop a practically usable universal apparatus of information representation due to their abstractness.

2.2. The Short History of Programming.

The fact that the general information theory has not been developed up till now by no way means that it could not be developed in principle. The true reason for its failure is the inability of the aforementioned theories in providing both *the uniform view of all kinds of information and the universal data representation tools*.

The answer to the not so rhetorical question -- what would such a theory and language be like -- can be found in programming, which used to be and still is the only source of practicable usable methods of semantically relevant formalizations of information. It may seem paradoxical that all aforementioned theories completely ignored the modern ideas developed in programming, but this disinterest has its roots in the recent history of computer science. The first theoreticians of programming were mathematicians and logicians who viewed it as a branch of mathematics. The result of their approach was an overly mathematized theory around the programming language. Some theoretical constructs like the formal language theory based on the ideas of the linguist Chomsky (Chomsky 1953, 1956, 1957) and theories of formal semantics have also been intensely scrutinized by information-related studies.

The break between computer science and information-related sciences occurred somewhere in the last decade of the twentieth century, when the software community quietly rejected high mathematized theories because of their ineffectiveness. Concentrating on the development of practically usable programming tools appeared to be much more efficient, but the essentially anti-theoretical way in which it was done basically excluded scientific discussions on the issue whatsoever. As a result, the information-related sciences were left out of the loop regarding new trends in computer science and the communication between these scientific domains stopped.

Yet more astounding was the failure of computer science to understand the reasons for its own fate.

Computer science in earlier times was an extremely lively discipline with the status of fundamental science. New ideas were intensively discussed in dozens of magazines and symposiums; multiple universities and firms created large numbers of new programming languages every year. All that at the end of eighties, when once feverish process of creating new languages lost its momentum bit by bit and nearly came to a complete standstill. The interest in new languages as well as the ambitions of their creators sharply declined.

From more than 2500 documented programming languages created in the last sixty years (O'Reilly Media 2004), only several dozen have been developed since the beginning of the nineties. Also, the goals of language developers underwent a change. New languages were no longer supposed to supersede their predecessors, as was often the case beforehand, but rather solve previously unknown programming tasks. A few remaining magazines and conferences discussing problems regarding the topic of programming language are rather tributes to the old glory supplying postgraduate students with publication opportunities than places for formulating and discussing new ideas.

The paradigm change in computer science was elevated by the radical transformation of the political and economic World Order brought about at the end of Cold War and rapid technical development.

The Cold War was, among other things, the golden age of science. Mutually perceived threats led to the immense boom in research and development with pioneering and innovative inventions and technologies created, fueled and sponsored by the military and its needs. Computer science was one of the main beneficiaries of this world order. At the end of the Cold War, research became increasingly more dependent on economic needs due to its privatization and as normally occurs with previously privileged branches, computer science was hit hard by the new reality.

The spreading PC revolution completely changed the needs software developers had to satisfy. Personal computers once considered expensive toys became full-fledged computers whose performance increased exponentially. They were readily available to all individuals instead of belonging to a small elitist circle of researchers; miniaturization and everyday requirements once again shifted the research focus away from high theory to practicability.

In addition, there was Moore's law accurately predicting the doubling of computer performance every 18-20 months. Faster hardware made sophisticated software solutions caused by scarce resources of the earlier decades of the computer era obsolete. In fact, the PC revolution actively contributed to the demise of economically unsustainable research teams that had been responsible for software developments in previous decades.

On the other hand, all these, doubtlessly significant events, were not able to stop the scientific research because the basic motive of this very research was just as reasonable in a new economic environment. New languages were created in order to solve the fundamental deficiencies of old languages and as long as these deficiencies existed, research had to continue. Consequently*, the only plausible explanation for stopping the eternal development process was fixing the problems that caused its prolongation*.

The behavior of the software industry exactly fits this explanation. One of the richest and most innovation-friendly industries ever, the software industry invested countless billions of dollars in literally everything but hardly showed any interest in examining new approaches to programming languages. The same is true for the open source community who developed huge numbers of freeware of very different functionality but paid astoundingly little attention to this area.

Another argument in favor of this explanation is the overwhelming efficiency of Moore's law. Though it is viewed as self-evident, it is not a necessary result in general. Just imagine how much less efficient the final impact of Moore's law would be if a compiler needed a tenfold increase in computer performance when doubling the size of the compiled program.

Similar restrictions however never surfaced in reality in spite of the size of the software installation packages going from less than one megabyte at the beginning of the nineties to several gigabytes today. This unprecedented scalability was achieved without changing the basic programming means. Twenty years ago, the main compiled programming language was C, now it is its pure superset — C++. The fact that the tremendous growth of the software size was implemented within the limits of the same language concept is the ultimate proof of the maturity of this concept.

Summing up the above, the older computer science became obsolete because the historical competition between different approaches to programming ended with the creation of the universal compiled programming language C++[3].

The development of programming languages started a few years after the appearance of the first digital computers because direct programming in machine code was exhausting

[3] a) In this work, C is viewed as a subset of C++ and not as a separate language. Accordingly, C++ and C/C++ are synonyms.
b) The term universal programming language is hardly used in modern day computer science because of the controversial discussions that arose long ago. Today when the final output of former developments is long established, the genuine intentions of then developers can be appreciated in a different way. It is now clear that the ills of programming tools of that time could only be cured with the help of a language which was powerful enough to be used as a single compiled programming language on a general purpose OS. A language that allows one to do so is referred to as a universal language.
c) Strictly speaking, the links between the rise of C++ and the decline of the old computer science are rather indirect. The practically oriented part of the old computer science, concerned with the effective compilation of programming languages, suffered a loss of interest in its methods because of increasing hardware performance. But the final blow to its perspectives was the development of so called Single State Assignment form (SSA)(Cytron et al. 1991) that allowed fast optimization of big programs and made any kind of further scientific research unnecessary. The language independent SSA could be (and was actually) used for the implementation of distinct programming languages, but as a result it only strengthened the position of C++ by effectively supporting the unique scalability of the latter. Which once more emphasizes the superiority of C++.

and slow[4]. The first implemented programming languages were primitive interpreters like Short Code appeared in UNIVAC in 1952 (Schmitt 1988) and Speedcoding proposed for IBM-701 in 1953 (Backus 1954). The first working assemblers appeared at the same time further simplified programming by allowing the use of symbolic names instead of physical addresses and numerical codes of machine commands.

The first high level programming language FORTRAN (Formula Translator) appeared in 1956 and for the first time allowed programs to be written oriented on its logic and not on the implementation of machine code. IPL (Information Processing Language), Lisp (LISt Processing), COBOL (Common Business Oriented Language) and others closely followed. As early as 1957 it had already become clear that the problem of machine code programming was obsolete and the real problem hindering effective programming was the growing number of specialized programming languages.

No wonder the quest for a universal programming language was defined early on as the topmost priority of computer science. But what is a universal programming language? The answer to this question was not clear at the time, in particular because there were distinct interpretations of the notion of *universality*.

- The perfectly impracticable universality of the Turing machine was formally defined by the British mathematician Alan Turing in 1936 (Hodges 1983). A Turing machine was a logical device that could scan one square at a time (either blank or containing a symbol) on a paper tape. Depending on the symbol read from a particular square, the machine would change its status and/or move the tape backward or forward to erase a symbol or to print a new one.

  This machine was considered by the theorists of programming as a real universal device, because it could presumably execute every existing algorithm. The fact that no one ever tried to build such a machine due to its absolute uselessness was never able to impress the elevated minds. In fact, this far-from-the-real-world understanding of universality only succeeded in making the issue more confusing without any positive impact. It also contributed to the demise of the former computer science because the science assigning the distinguishing role to such an unusable interpretation was increasingly detached from reality and practical needs, disregarding topical pressing demands.

- Another idealistic understanding of universality viewed mathematics as the philosopher's stone of programming. According to the proponents of this approach, it was only necessary to find the proper formulas and every problem of programming

[4] The review of programming languages' history was created using the sources (O'Reilly Media 2004; Sammet 1969) (Sebesta 1993; Pratt and Zelkowitz 2000; Ritchie 1993)

could be effectively solved. Over time, this approach developed to the idea of declarative languages, which allow the expression of the logic of a computation without describing its control flow. Declarative programming often considers programs as theories expressed by means of formal logic and computations as deductions in that logic space.

Two kinds of declarative programming were once considered extremely promising. One was functional programming that treats computation as the evaluation of mathematical functions and the other, logic programming, which is based on mathematical logic. The most prominent programming language of this type is Prolog. While declarative languages generated a lot of noise in computer science, their impact on the programming practice was hardly noticeable.

- Two close to reality understandings of universality were actually used in practical programming. Here is how they were characterized in 1968 by Jean Sammet (Sammet 1969, 723) in her outstanding study of programming languages.

  "It is no accident that the development of a specially created *universal programming language* is omitted from this chapter. It is my firm opinion that not only is such a goal unachievable in the foreseeable future, but it is not even a desirable objective. It would force regimentation of an undesirable und impractical kind and either would prevent progress or, alternatively, would surely lead to deviations. However, the possibility of a single programming language with powerful enough features for self-extension to transform it into *any desired form* is interesting to consider. (by *any desired form*, I mean all the languages in this book, plus any other which are developed subsequently) The techniques for this development are clearly unknown currently, but they could conceivably be found in the future".

  The notion designated in this citation as *universal programming language* is the concept of an universal high-level programming language. Such a language is designed to try to give programmers everything they could possibly want already built into the language. This understanding dominated programming several decades before it finally succumbed to a middle-level programming language (*a single programming language* in the citation), which used a minimal set of control and data-manipulation statements allowing high-level constructs to be defined.

The first project of the universal programming language was Algol-58 (ALGOrithmical Language), originally defined by the Zurich-based International Committee as the Esperanto of the Computing World in 1957. Due to its complex constructs and inexact specifications, Algol-58 and its successor Algol-60 were not implemented until the end of the sixties, but they sparked the creation of other languages such as NELIAC (Navy Electronics Laboratory International ALGOL Compiler, 1959), MAD (Michigan Algorithm

Decoder, 1960) and JOVIAL (Jule’s Own Version of IAL, first implementation in 1961), the latter one being the first attempt to design a programming language that covered several application areas.

The more successful attempt of a general purpose language was PL/1 (Programming Language number One, originally NPL – New Programming Language, 1964) which was considered as a substitute for the majority of programming languages existing at that time, including FORTRAN, COBOL, ALGOL and JOVIAL. Despite being actively used by the software community, PL/1 fell short as a universal language mostly because of its over-complexity and thus the creation of new specialized languages continued.

The next attempt in developing the universal programming language was Algol-68, initially defined in 1968. The language was overly complex and seldom used.

The final and most expensive effort to create a universal programming language was Ada whose development started at the request of the USA’s Department of Defense in 1974. The department struggled to reduce the number of programming languages used for its projects, which reached around 450 at the time, despite none of them being actually suited for the department’s purposes. The result was an overly complex programming language first implemented in 1983. Similar to its predecessors Ada was not very successful and is now used limitedly mainly in military projects where it originated from.

The most interesting paradox was that the first really universal programming language was already implemented even before the start of Ada’s design efforts. The language C, created by Dennis Ritchie between the years 1969 and 1972 was “a system implementation language for the nascent Unix operating system. Derived from the typeless language BCPL, it evolved a type structure; created on a tiny machine as a tool to improve a meager programming environment” (Ritchie 1993). C was called a middle level programming language because it lacked many features characteristic of high-level languages. Though namely its parsimony was the true basis of its advantageous characteristics since the missing capacities were the true reason for the poor performance of high-level programming languages.

C provided such effective low-level access to memory that assembler programming became obsolete. Its capabilities also encouraged machine-independent programming. A standards-compliant and portably written C program can be compiled for a wide variety of computer platforms and operating systems with little to no change in its source code.

In 1983 Bjarne Stroustrup, then a researcher at Bell Labs, implemented the object-oriented extension of C called C++ (originally C with Classes)(Ellis and Stroustrup 1990) and the everlasting run for the new languages gradually lost its appeal. The universal compiled programming language was born!

2.3. The TMI View of the World

The conceptual system of TMI is based on the extension of notational system of C++, which together with its ancestor-subset C, constitutes the base of virtually all complex software as operating systems, programming software (compilers, interpreters, linkers, debuggers) as well as complex application software for various purposes. The unflappable dominance of C++ is in essence the ultimate proof of universality, reliability and consistency of the conceptual system of this language.

In the world of programming languages universality is however not absolute. C++ was able to replace other imperative compiled programming languages like COBOL, FORTRAN, PL/1, Ada, Pascal, though not the interpreter-based programming languages such as Java, JavaScript, PHP, Perl, Basic, command shell and so on. Though C++ has steadily demonstrated its influence also in foreign programming domains. Most of the object-oriented programming languages designed in the last 20 years are based on the C/C++ like syntax, in spite the fact that this syntax is more complex than that of Pascal.

C++ played a decisive role in establishing object-oriented (OO) programming[5], which was developed as the dominant programming methodology in the 1990's. C++ was neither the first object-oriented programming language (which was Smalltalk-80) nor the only object-oriented extension of an existing language. Object-oriented features have been added to many existing languages including Ada, FORTRAN, Pascal and others but none of them have survived. Adding these features to languages that were not initially designed for them often led to problems with compatibility and maintainability of code.

C++ has avoided these problems because it was originally built as the minimally possible extension to already minimal C. The minimalist structure made this language extremely efficient and actually discouraged further attempts at producing plausible alternatives, except for languages like the programming language D (Alexandrescu 2010) claiming itself to be the better C++.

The current state of information-related sciences is in many ways similar to programming in its early decades. Like the latter, the former also uses a lot of specialized languages. Like the latter, the former understands that a universal language is possible (a natural language delivers an ultimate example thereof). Like the latter, the former possess distinct

[5] Basically, there are two understandings of object-orientation. One is that an object-oriented language has to support strict code-composing rules, like packing all data in classes and all algorithms in methods, disallowing direct access to the class members, prohibiting pointers, go to operators and so on. Another understanding is that an object-oriented language allows the free combination of object-oriented code with conventional imperative programming. The latter understanding, embodied by C++, is also shared in this work.

interpretations of universality. The difference is that the final outcome in programming (C++) is already known at this time and can also be used by information-related sciences for developing truly universal methods of information representation.

Of course, C++ is very far from being the universal representation language required by a general information theory. It is a programming tool possessing this strange thing called implementation seemingly missing by the natural languages. Its expression power is restricted to the representation of bit sequences allocated in the computer memory and the only expressible operations are sequentially organized manipulations with these bits. On the other hand, C++ as well as the methodology of object-orientation are unquestionable owners of the patent for universality (even if only in their restricted representation domain) and that is the only thing, which matters in this case.

Henceforth the approach of TMI consists of extending universal methods of information formalization developed in programming *outside of their original domain*.

***

According to TMI every real or imaginary entity can be viewed as an object whose behavior can be expressed in the manner of programming algorithms. For example, the algorithm of the Earth consists of the cyclical movement around the Sun and the rotation around its axis each day. In the same way, the structure of the Sun, Earth and their algorithms can be described in English they can also be described in the universal representation language ***T***, which expresses these algorithms with the help of object-oriented notation based on the conceptual system of C++.

In essence, ***T*** can be considered as the extension of the expressing abilities of C++ linguistic signs to representation of entities exceeding the C++ representation world. The only entities, which can be referenced in C++ are bit sequences allocated in the computer storage and the only operations are sequentially organized manipulations with these sequences. Any entities exceeding these objects and these algorithms cannot be expressed in this language. This different with ***T***, which allows composing miscellaneous discourses (exactly in the way of a natural language).

The definition of ***T*** is not associated with any particular implementation and consists of the definition of the language grammar, usage rules and core vocabulary in the way of a natural language like English or German. Notions of ***T*** like class, instance, procedure, variable, parameter and so on stand for nouns, verbs, adjectives, pronouns and other parts of speech.

T allows implementations with semantics of existing programming languages, like compiled languages (C++, Pascal) or interpreted languages (like Java, Basic, Command

Shell). Ultimately, it allows the creation of a monolingual communication environment in which it is used as the only language for both programming miscellaneous tasks and composing diverse non-executable formal documentation.

***

While the term *unambiguousness* can be understood in a number of very distinct ways, here is its interpretation used throughout this work.

In TMI, an unambiguous text is a text that allows the only meaning in the given context. Thus, if there is a set of objects called `objset`, every element of which can be identified by some unique number, the text "`objset[12345]`" unambiguously refers a single item in this set.

This text will become ambiguous however if distinct items share the same identifier or there is more than one set with the name `objset`. Ambiguous texts are considered to be incomplete descriptions missing the necessary characteristics of the subject in question. To the contrary, an unambiguous representation of some entity has to contain as many of the entity's characteristics as necessary in order to narrow down the number of represented meanings to only one.

T is able to produce unambiguous representations by enabling the unlimited addition of missing characteristics. The characteristics can be of various kinds like the location of the entity in time or space; the objects, events, states associated with this entity; the real or imaginable world to which this entity belongs and so forth.

***

Language and semantics. The traditional understanding of relations between a language and semantics (meaning) conceives the latter as a part of the former. This interpretation follows from both the history of the study of semantics, which was originally developed as a subfield of linguistics and the purpose of this study, which was traditionally understood as "the study of the meaning of linguistic signs" ["Linguistics", in: Encarta].

Superior entities in TMI are knowing objects (communicators) using a language for exchanging semantics with each other. Communicators are persons, computers or any other entities that possess some semantics (knowledge) and want to exchange with other communicators.

The relationship between semantics and a language is equivalent to that between the freight exchanged by some senders/receivers and delivering services transporting it to the required locations. While a delivery service may be excellent at moving goods from point **A** to point **B**, it generally has no competence outside of that, because it neither produces nor uses the transported freight. The only way to understand the transported freight consists of studying its senders and receivers.

The approach of TMI consists of the explicit definition of communicators participating in every communication. Communicators are primal language objects that exchange information with the help of language code. Their structures and algorithms are quite similar to objects and algorithms specified in a programming language like C++, but differing from the latter they cannot be represented in an executable code.

3. The Information Theory

3.1. The General Model of Meaningful Information

3.1.1. Definition of Information

According to the standard theory of cosmology, literally everything – matter, space and time – started with the Big Bang. In the beginning, all matter and energy, which made up the universe was squeezed into an infinitely hot, dense, unstructured singularity. It then started to expand, became more structured, the fundamental forces were divided, matter formed to atom nuclei, leptons and molecules. The content separated into distinct components, parts of which were more stable than others and so preserved their form and size over long periods of time.

So was formed our universe, which we can consider as a super heap consisting of stable objects of different levels of matter organization and their heaps. The process of universe formation and evolution consists of permanent births, deaths, and changes to all entities of the heap. A stable object here is understood as a three dimensional item with mass, which is reasonably steady, has a location or position in space at any moment, and which can be changed by exerting force. The stable objects of the lower levels of the matter organization are inanimate physical bodies like stars, planets, stones, molecules, atoms and subatomic elements like hadrons, leptons etc. and those of the upper levels are living organisms of various complexity levels starting from unicellular bacteria up to human beings.

Under the influence of force, a stable object changes its speed, movement direction, starts/stops moving if it is exposed to the physical (energetic) influence of another object or process (like one object hitting another object and changing its movement etc.) and is distorted. A body in rest can come into motion if the balance of forces maintaining its immobility is disturbed, e.g. a bridge is stable because its weight is balanced by the counter-pressure of its pillars and it fails if one of the pillars is damaged.

A general characteristic of force-induced changes is that the effectual change occurs exclusively because of the energy produced or withdrawn by the causal action (as in the case of the falling bridge).

Stable objects are not really static but rather dynamic combinations of elements. Electrons revolve around atomic nuclei, quarks exchange gluons in hadrons; molecules are involved in chemical reactions in the body of cosmic objects and cellular organisms. Stated more precisely, stable objects are dynamic systems that are better at adapting to the environmental conditions around them and therefore can retain their form and stability. This ability to adapt means they are able to resist external and internal influences by either ignoring them or changing in order to counteract the influence. Since the ability to resist is limited, a stable object will collapse if the force influencing it exceeds a certain threshold, and it will be changed.

The life of a stable object can last from a fraction of a second to billions of years and during this time constantly moving and/or changing. The sequence of changes and movements of a stable object is defined in this work as the object's algorithm that can be represented with the help of various means like pictures, schemes and texts composed in various languages.

The subject to be considered here is the causal relations between the changes of stable objects and their environments. The phenomenon of causality, famously characterized as the cement of the universe by David Hume (Davis, Shrobe, and Szolovits 1993), includes many miscellaneous causal relationships, but we will limit ourselves to changes occurring in objects under various influences in a Newtonian world.

According to the Random House Unabridged Dictionary ("RHUD" 2002), the term causality denotes "a necessary relationship between one event (called cause) and another event (called effect) which is the direct consequence (result) of the first."

The term change is defined in the same dictionary as "to make the form, nature, content, future course, etc., of (something) different from what it is or from what it would be if left alone".

The axiom of this work is that all causal changes in the real world are based on only four distinct schemes: two primitive causal relations occurring under the influence of physical forces and two complex ones. Changed entities are either complex stable objects including several components or systems thereof.

3.1.2. Change I driven by an Internal Force (internal change)

Changes of this kind are independent from the external environment of a changed object or at least considered to be independent on the level of the analysis of the object. A stable object alters its form, position or structure under the force produced by its internal processes. Such a change is the most frequent case of alterations among living organisms. Growth of living creatures, human mental processes or cellular division can be cited as examples.

Non-living physical entities are also changed in this way, especially those consisting of a dynamically balanced set of physical processes such as cosmic stars. Stable atoms disintegrate due to spontaneous radioactivity, which can take a very long time. This simplest causal mechanism occurring within a single stable object is designated here as internal change.

Internal change will be represented with the help of a horizontal rectangle that is split along the x-axis into two parts. The upper half refers to the changed entity and the bottom half describes the change to this entity. Because this object changes spontaneously without any apparent external force or influence, there is only one rectangle and no other objects or arrow directions contained in this graphic.

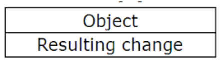

Figure 3-1: Spontaneous change

3.1.3. Change II driven by the External Force (forced change)

The second basic kind of causal relation consists of changes that occur as a result of externally working forces. According to Britannica force is “any action that tends to maintain or alter the motion of a body or to distort it” (“Force” in *Britannica Encyclopedia Ultimate Reference Suite* 2009).

The mechanism of this change is simple: the influence of a force alters the affected object or its movement: A cosmic body changes its orbit due to the influence of gravity produced by some external entity. A bullet accelerates because of the powder explosion in the cartridge; glass breaks when it falls to the floor etc. This form of causal mechanism is called forced change here.

Forced changes represent the simplest form of a causal relation between two objects, the first of which — a causal action of force — delivers energy causing the effectual change of a stable object. Forced change is represented as follows:

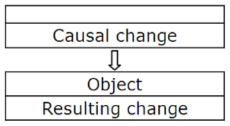

Figure 3-2: Externally driven change

The empty part of the upper rectangle illustrates the fact that a causal change can have an arbitrary origin. The arrow designates the direction of force. The vertical order of rectangles represents their relationship in time.

3.1.4. Change III activated by External Force (activated change)

This and the following causal mechanism are based on the sequence of primitive changes described above. The entities being subject to change are complex stable objects with several components and internal energy sources or systems consisting of several components and energy source(s). Furthermore, the changed objects are either organic organisms or inorganic systems of artificial origin. Inorganic objects of natural origin are too primitive for such complex behavior.

The activated change consists of changing a stable object (system) by external non-forceful influence. A changed object (system) has to possess (or have access to) an energy source and an externally controlled lock (switch), which opens and closes the energy flow produced/delivered by the energy source.

A typical example of this kind is an electric light activated by an electrical button or lever when someone (something) switches it on and the electrical bulb lights up. The sequence of actions includes the following four activities:

(1) external force presses the button;

(2) the button closes the electrical circuit thereby unlocking the energy source;

(3) electricity flows to the bulb;

(4) bulb lights up.

The general graphical schema of this change is as follows:

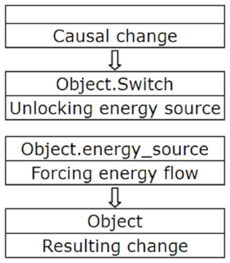

Figure 3-3: Externally activated change

The difference between change II and change III is the addition of the two externally activated steps two and three. The lack of an arrow between them shows the missing energetic influence between the respective changes. This change is thus a sort of combination between change I and change II with the internal change being triggered by an external force.

The thing activating an internal change by external influence is referred to in this work as an activating switch or an activator.

Examples among biological objects are unicellular organisms registering the change of the environment by their sensors and changing in response the process of their metabolism. Components built on the principle of activated changes are integral parts of any technical system like electronic circuits.

A more elaborate example would be a traffic light at a pedestrian crossing (crosswalk). A person approaches a crosswalk, pushes a button, the traffic light changes to red and the crosswalk sign flashes green. The only work a person has to do is push a button and the rest is done by the internal component(s) and energy source(s).

The specifics of this type of causal relation are the energetic independence between the activating action and the resulting change which allows various activation-result pairs with the same internal mechanism. The energetic independence radically increases the survival chances and adaptability of systems implementing it.

Let us imagine there is a sea lagoon fed by an intermittent river. Some microorganisms, whose nutrition is delivered by the river's water, have been able to adapt to the changeable conditions by slowing its metabolism after the level of oxygen in the water drops below a critical point. The internal mechanism registering the level of oxygen is the typical activator changing the behavior pattern of these microorganisms in result of the alternation of the external environment.

If some species of this microorganism are carried by currents into the open sea, they encounter another changeable food source consisting of unsteady sea currents of very salty water. Surviving in these conditions requires another activator, which would regulate the microorganism's metabolism by reacting to the change in water salinity, while activator-
controlled basic structures can be left unchanged.

From the viewpoint of evolution, it is a much more efficient way of development than the development of a completely new microorganism, which would be necessary if the activating switches were not involved in the evolutional process.

3.1.5. Change IV communicated by External Force (communicated change)

The mechanism of activated change provides an essential part of flexibility relative to primitive causal mechanisms I and II, on the other hand, it also has its restrictions convincingly demonstrated by the traffic light example. In the form delineated above, these traffic lights are hardly usable on a busy street, because many pedestrians who want to cross the street will actually block the traffic by persistently pushing the crosswalk button.

In order to solve this problem our traffic light needs to be able to define the switching moment on its own, i.e. by changing the cause-effect direction. Assume that a person does not push the button, but the button will be periodically (e.g. once every 3 minutes) pushed against the finger of a person presumably pressing the button. If the finger blocks the button, the crosswalk lights switches to green for the pedestrians and red for cars. Otherwise, the color of the traffic light remains the same.

Certainly, this is not a very convenient schema for traffic lights and it does not give pedestrians utter satisfaction either. A more efficient solution to this problem consists of augmenting the switching schema by adding one more element – a passive mediator object (in this case a button together with associated components), which can be pressed by a pedestrian and queried periodically by a traffic light. By pushing such a button, the pedestrian only switches the button to another state which can be ascertained by the traffic light during the querying process. If the button is set, the traffic light unsets it and then switches the crosswalk light to green for the pedestrians.

This example demonstrates the work of information, which is a state of some passive object set by some actor and queried by some reactor in order to produce some resulting action. An actor has to set at least one state and a reactor has to distinguish between at least two states (the absolute minimum is a dichotomous variable – the state set by an actor and the lack thereof).

The following definitions will be used throughout this work:

- The mediating object is a variable. Wherefrom follows a yet even shorter definition of information being conceived as a value of a variable used in an algorithm.
- Communication is the process of interaction between an actor setting a variable's value and a reactor identifying it and performing these or other actions as the result.
- An actor is designated as an Information Setting Entity (ISE) and a reactor as an Information Driven Entity (IDE).
- The resulting change constitutes the semantics of information.

Relative to the activated change, this kind of causal relation has been enhanced by the addition of three new steps inserted after the causal change.

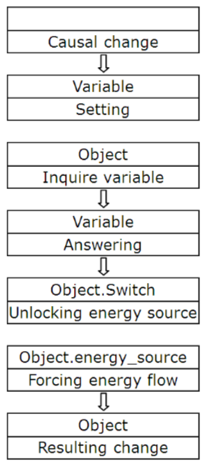

Figure 3-4: Communicated change

In this work, a variable is not considered a mathematical construction, but an object in the discernible world. Since the features of communicators and communications can differ, the structure of the variable can differ, too. Thus, in implementation where the crosswalk light directly differentiates between pressed and non-pressed buttons, the variable consists of the complete button mechanism. In alternative implementations where the button mechanism is connected to the input port of the traffic light's internal processor the mediated variable is the input computer port and the button mechanism is an auxiliary appliance used by a pedestrian for setting this variable.

It is not required that a variable can be switched unlimitedly between its states. A variable is considered to be constant if it cannot change during its whole life span as e.g. a printed letter, which is essentially a picture painted on some surface. Other special cases are variables that can be changed irreversibly, i.e. only once, e.g., an undamaged pencil can be used as one sign and a broken pencil as another sign.

This understanding of information complies with the one actually applied in programming. The variables in programming are restricted to bits or bit sequences and the only two fundamental operations with variables are set and query, whereas all others are based on these two operations.

Another restriction in programming is that the actor and the reactor are usually the same, so the communication occurs not between different objects but between different states of the same computer.

3.1.6. Summary of Causal Relations

All four types of changes are given in Figure 3-5. Each rectangle contains the stable object (if present) and its change. The order from top to bottom represents the timely flow. The arrows represent a force being exerted.

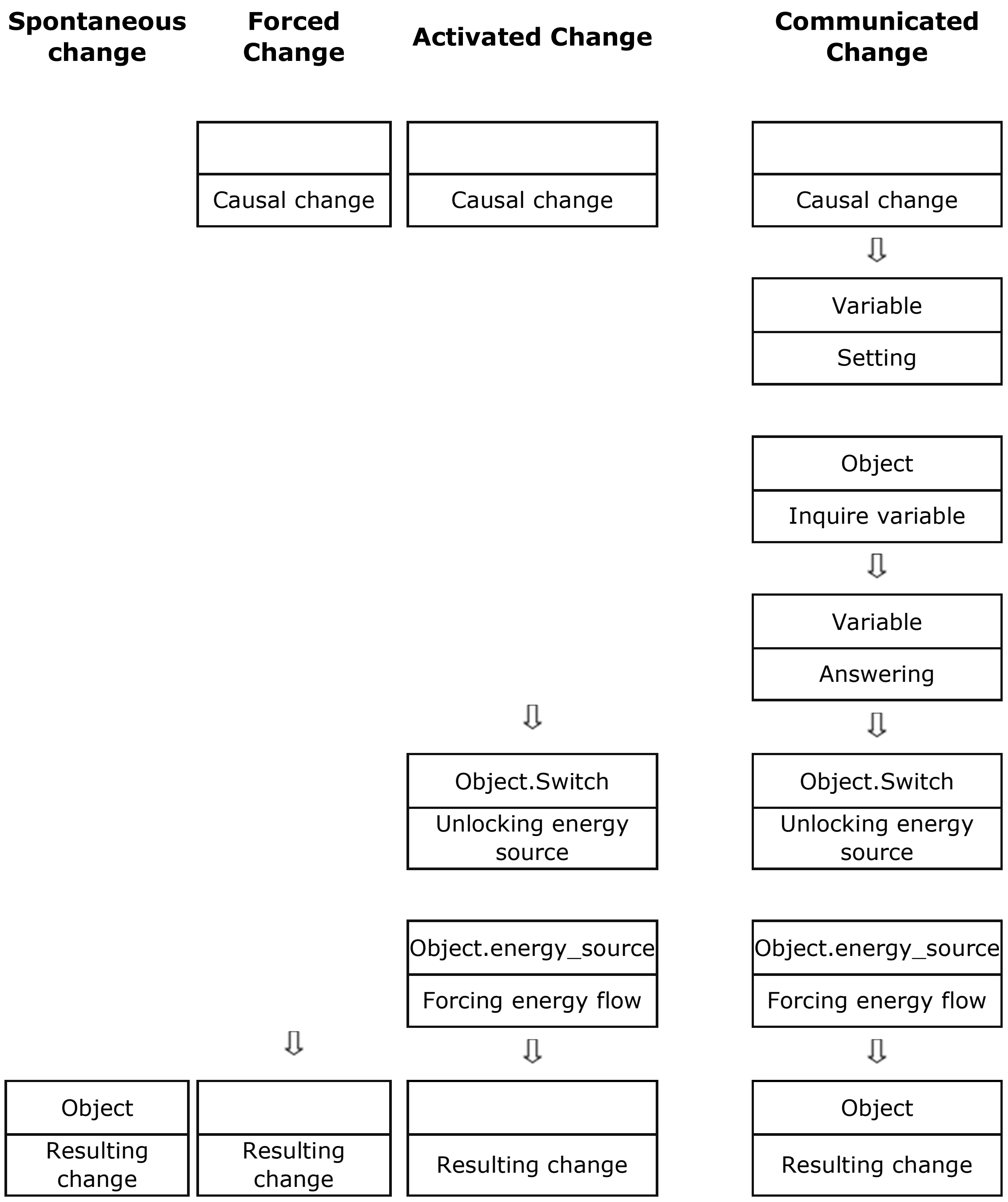

Figure 3-5

3.1.7. Information Atoms

In general, there are three groups of processes associated with communication: information production (setting activity), transmission of information from one place to another and use (resulting activity) as depicted in the following Figure 6.

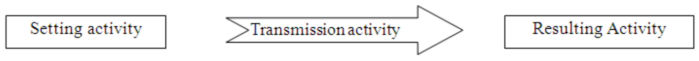

Figure 3-6: Information-related activities

The transmission process (which was the only subject of Claude Shannon's theory of information (Claude E. Shannon and Weaver 1948) can be reduced to nothing, as in the examples of the traffic light and microorganisms. In other cases, it can be very sophisticated and may include numerous steps, each of them producing its own information representations. The stages of the transmission process are designated in this work as information processing.

(1) By their nature, variables while differing from activators are also switches involved in two basic activities. The operation *set* switches a variable to some value, and the operation *query* investigates the actual value of a variable and changes the IDE by switching the inquiry mechanism.

The difference between an activator and a variable is functional not physical. A computer bit, the most primitive variable on the computer level, functions as an activator on the level of a memory chip implementing it. Another good example would be a neuron in the brain acting as a variable, whereas a motor neuron connecting the brain with a muscle is an activator.

(2) Activators and variables represent two levels of information carriers. The activators are low-level entities which unite both set and query in the only action while the variables represent the high-level information processing in which these two functions are separated from each other. Essential differences between these two forms of information application are:

- While an activating switch immediately causes the resulting change, the setting of a variable is detached in time from the querying result.
- A single variable can cause many changes in miscellaneous ICE algorithms (see below under point 6.) versus the only change caused by an activator;
- Many variables can be queried with the help of a single inquiring mechanism, which is essentially a switch.

(3) The term varier is used in this work as the most abstract designation of an information carrier, which can be either an activator switch or a variable. Variers of naturally developed ICEs from the level of cnidarians (corals, jellyfish) are neurons.

In this work, a neuron is considered an activating switch changing between no-output and output states by reason of the relevant external signal for this neuron. Depending on the type of input signal, neurons are classified as changeable and non-changeable. An example of a non-changeable neuron is a motor neuron, which transmits impulses from a central area of the nervous system to an effector, such as a muscle. An example of a changeable neuron is a brain neuron, whose activating signal can be reprogrammed to respond to a number of signals(Best 1990).

(4) A stable object can only be considered an IDE if it contains at least one varier. The communication between an environment and an IDE occurs with the help of sensors, which are essentially activators. The architecture of an IDE containing only variables but no activators is physically possible but does not comply with IDE principles because such an entity is completely independent from the environment.

(5) Of both IDE and ISE, only the first inherently possesses the information related functionality whereas an ISE can be an arbitrary phenomenon of the organic or inorganic world with or without information related abilities. An IDE with the mediated varier as its interface constitutes the kernel part of the communication environment, which in general can get information from miscellaneous ISEs. An object possessing IDE capabilities with or without ISE capabilities is designated as an Information Capable Entity (ICE).

(6) As long as the same cause produces the same effect, the nature and structure of (a) varier(s) is irrelevant. If a certain lagoon is periodically filled with water without sufficient nutrients, the organisms living there can change their metabolism by reacting to miscellaneous environmental parameters such as water salinity, water temperature or water density with the help of the distinct inquiring mechanisms. The same is true in the case of artificial devices. The traffic light will function in the same way with miscellaneously designed switching buttons.

3.2. Information Capable Entities (ICE)

3.2.1. ICE Classification

All currently known ICEs were developed on Earth as a result of either the evolutionary process or human engineering. However, their features and structures are very different. Naturally developed unicellular and multi-cellular organic organisms were followed by social organisms and artificial devices created by the most complex natural ICEs — human beings. Known ICEs can be separated into the following classes and subclasses based on certain fundamental characteristics:

(1) Biological organisms

- Unicellular microorganisms — the simplest form of naturally occurring ICEs
- Multi-cellular organisms from non-structured sponges up to human beings
- Social organisms ranging from primitive organizations like ant colonies up to various created human societies and states.

(2) Artificial objects and systems

- Non-computer artificial objects and systems. These are entities, which use the simplest forms of information and are normally unable to produce information. Among them are all devices powered by electrical currents equipped by at least one controlling switch. Such devices also possess various sensors accepting these or other information.
- Computers and non-computer systems containing one or more embedded processors (everything from dishwashers up to cars, airplanes etc.)
- Components of all aforementioned entities, possessing ICE functionality.
- Software entities.

The only characteristic common to all ICEs is that they have almost nothing in common. They are studied by distinctly different fields of science, each using its own conceptual apparatus which is hardly compatible with those of the other sciences. Furthermore, ICEs are not necessarily confined to any specific location in space as do social organisms or are seemingly immaterial in the way of software entities.

The task of defining a common denominator for all these systems is by far the most complex of all TMI undertakings. The solution consists of interpreting these systems as Information Capable Entities whose structure and functionality are defined in terms of their abilities for the use and production of information. Two ICE classes are worthy of special consideration.

First, software entities differ from other ICE classes in that they are not independent. The software is the part of the computers where it runs and cannot be understood without context of its computing environment. But on the other side, they are the only kind of ICE which is effectively expressed by formal (programming) languages and hence they are widely used in this work for the purpose of demonstrating IDE and ISE characteristics.

Another very specific kind of ICE is human beings, who are the most complex ICEs of natural origin. In this work, human beings are viewed as biological computer-controlled-devices following the idea first formulated by John Lilly in 1968 "Programming and Metaprogramming in the Human Biocomputer" (Lilly 1968). Despite the unbelievable progress in computing occurred in the last forty years Lilly's ideas have not lost their relevance. Listed below are excerpts from his book(1968, 20,21):

- The human brain is assumed to be an immense biocomputer, several thousands of times larger than any constructed by Man from non-biological components by 1965. The brain is defined as the visible palpable living set of structures to be included in the human computer.
- The numbers of neurons in the human brain are variously estimated at 13 billion with approximately five times that many glial cells. This computer operates continuously throughout all of its parts and does literally millions of computations in parallel simultaneously. It has approximately two million visual inputs and one hundred thousand acoustic inputs. It is hard to compare the operations of such a magnificent computer to any artificial ones existing today because of its extremely advanced and sophisticated construction.
- The computer has a very large memory storage and controls hundreds of thousands of outputs in a coordinated and programmed fashion.
- Certain programs are built-in, within the difficult-to-modify parts of the (macro and micro) structure of the brain itself. Other programs are acquirable throughout life.
- The human computer has stored program properties. A stored program is a set of instructions which is placed in the memory storage system of the computer and which controls the computer when orders are given for that program to be activated. The activator can either be another system within the same computer, or someone (something) outside the computer.

The view of human beings as devices controlled by a biocomputer allows them to be compared to artificial devices controlled by digital computers which, amongst others, enable the description of biocomputers in terms traditionally used for digital computers like memory, input and output ports, code and data, algorithms run on the computer etc.

While both kinds of computers are similar in many characteristics, they are completely different in memory organization and the order of command execution. Digital computers have a memory consisting of sequences of bits organized in bytes and controlled by a central processing unit processing commands put in the computer memory, whereas biocomputers are built of neurons which, at least at first glance, function on completely different principles and do not have anything in common with the CPU of digital computers.

3.2.2. Biological ICE

The view of evolution detailed below does not coincide with Darwin’s theory, which is strongly based on biological development. The subject of the proposed view is the general algorithms of development regardless of whether biological organisms or artificial systems. The process of evolution is seen as the process of developing biological entities starting from the simplest biological organisms up to human beings. Evolution started on a still hot Earth, where it first produced systems of non-organic molecules under the influence of external factors united with organic molecules whose interaction once gave birth to the first unicellular organism and so on.

The objects controlled by the information influence differ in their inner organizations. The simplest ICEs are single-level objects with one or several sensors. These are systems like the aforementioned traffic light or such unicellular organisms, which can choose the moment to react by themselves.

The next level of complexity is programmed objects, which in addition to externally accessible information carriers, also possess internal variables whose values are set during the process of creating the ICEs (hard-coding) or during their lifetime (soft-coding).

Evolutionally hard-coded systems preceded soft-coded ones. Thus, all primitive organisms without conditional reflexes such as insects, amphibians, and so on mainly relate to this kind. Soft-coded organisms are those with conditional reflexes and are mainly mammals that can learn and in this way better adapt to changing environmental conditions.

The following enumeration of stages of ICE evolution—from the simplest to the most complex — is in no way complete. Its only purpose is to provide a general description of the different kinds of ICEs without going into detail. It describes the types of information-related subsystems of naturally developed ICEs, but all these types can also be demonstrated on examples of artificial ICEs, because they are much simpler and transparent than natural ones.

3.2.2.1. One Varier ICE

The simplest ICE possesses either one or several independent sensors supplying information from the environment. The example of such an ICE is a unicellular organism (as in 3.1.4) which can react to a lowering of the oxygen level by freezing its metabolism, and vice versa.

3.2.2.2. ICE with Hierarchy of Variers

Just imagine the species of microorganism described in 3.1.4 being transported by sea currents from a lagoon to the open sea and back. In order to survive they need to adapt themselves to changing environmental conditions, activating their metabolism with increasing oxygen levels in the lagoon as well as increasing salinity in the open sea.

In order to do so, they need a third sensor recognizing the environment. Because water in lagoon is much warmer than that in the open sea, such a sensor can recognize the environment by measuring the water temperature.

In this case, there is a hierarchy of activators with the upper sensor recognizing the environment and two subdued sensors recognizing the state of the respective environment and functioning only after being activated by the upper sensor.

3.2.2.3. Programmable ICE

Programming is normally associated with modern digital computers, when, in fact, its history can be traced back to medieval times. The earliest known programmable machines (i.e. machines whose behavior can be controlled and predicted with a set of instructions) were Al-Jazari's programmable Automata in 1206. One of Al-Jazari's robots was originally a boat with four automated musicians that floated on a lake to entertain guests at royal parties. Programming this mechanism's behavior meant placing pegs and cams into a wooden drum at specific locations. These would then bump into little levers that operated a percussion instrument. The result was a small drummer playing various rhythms and drum patterns(Heslin 2014).

Another example is the Jacquard loom, which was invented by Joseph Marie Jacquard in 1801 (Heslin 2014). The loom simplified the process of manufacturing textiles with complex patterns. It had holes punched in a pasteboard, each row corresponding to one row of the design. Multiple rows of holes were punched on each card and the many cards that composed the design of the textile were strung together in order.

An example of the simplest programmed ICE is a modification to the above defined three-varier microorganism, which lives its complete life in one environment. The environment of such a microorganism can be checked only once at the moment of its birth by measuring the water temperature and then storing the measurement in the internal variable. Instead of constantly measuring the water temperature, it only has to evaluate the state of its internal variable, which is in actual fact the simplest program controlling the behavior of such a microorganism.

This kind of program was probably not used very often at the beginning of organic evolution. The primitive programs were molded in neurons controlling the movements in multi-cellular organisms. Thus, even the simplest flying insect can do a lot of maneuvering during flight, which requires extended programs to control the movement of its body parts and process the information returned with the help of miscellaneous external sensors.

3.2.2.4. ICE with Base Learning

Soft-coded organisms are those with conditioned reflexes. Examples of primitive organisms using this feature are bees, which can remember the source of nectar and communicate it to other bees. Mammals possess an advanced form of this mechanism and are able to learn complex things and thus better adapt to changing environmental conditions.

3.2.2.5. Passively Adaptive ICE

As soon as organisms became complex enough they started to optimize the learning process by imitating the behavior of their parents and other associates. In order to do that, they needed a way to establish the links between themselves and the associates. In the simplest case, this can occur as a result of inborn knowledge.

An example of such a link was demonstrated by the Austrian zoologist Konrad Lorenz in his famous experiment with newly hatched ducklings. He appeared before the newly hatched ducklings and imitated a mother duck's quacking sounds, after which the young hatchlings regarded him as their mother and thus followed him("Lorenz, Konrad" 2009). This occurs soon after hatching because the young ducklings intuitively learn to follow someone who they identify as their parents. The process, which is called imprinting, involves visual and auditory stimuli from the parent object; thus eliciting a following response in the young that affects their subsequent adult behavior.

3.2.2.6. Actively Adaptive ICE

Animals that are more complex are able to acquire knowledge about themselves, which allows them to identify with other organisms having the same smell, the same appearance and same behavior. This is an ability that, e.g. great apes or pigs, exhibit.(Broom, Sena, and Moynihan 2009).

3.2.2.7. Language Able ICE

The use of external language for exchanging information in the manner human beings do.

3.2.3. Intellectual ICE: - Human Being as a Biocomputer-controlled Device

The model introduced in this section considers a human being to be a self-programming system consisting of a brain (the computer) with the rest of the human body controlled by this computer. This model is based on the view first formulated by John Lilly (Lilly 1968).

The model allows the expression of the complete information used by a human being, starting from the simplest perceptions and ranging to high-level intellectual functionality.

A formal view of a human being constructed in the sense of this model will be designated as Homo Informaticus (***HI***). The body of an HI is seen as a complex set of moving parts, each consisting of a muscle attached to a bone or an organ. A muscle executes one of two commands contract or relax, which in turn causes a movement of a bone or organ. Another important part of this system are the senses, which consist of multiple receptors functioning similar to the input ports of conventional computers.

Differing from sequential programs of conventional computers, those performed by the HI’s brain represent multithread algorithms consisting of synchronized executions of many programmed units intermixed with perceptions from various senses. For example, the process of picking up a toy by a baby includes distinct synchronized sub-processes such as moving the baby’s eyeballs, neck, hands, arms, fingers as well as the changes of states of multiple visual and cutaneous receptors.

The process of self-programming an HI being starts immediately after birth and continues until death. This view also includes a detailed description of the self-programming algorithms as well as the ways in which these programs are used.

Knowledge of various kinds (skills, beliefs, concepts) is represented by the programming code and data (perceptions). The intellectual operations performed by the HI’s brain are interpreted as manipulations with the code of these programs, and a natural language is considered a system that enables programs to be exchanged between various individuals.

The tacit knowledge (father called experience knowledge) is the low-level knowledge acquired during the direct contact of HI with the immediate environment. The conceptual knowledge, which can be expressed with the help of a natural language, creates the level of communicated knowledge (information). This kind of knowledge consists of references to the experienced knowledge, changes and their structures. A change is the sequence of states of the same external object produced by internal comparison algorithms from the information delivered by HI’s senses.

An HI is essentially an abstraction concentrated on the information processing without thorough representation of the complete organism. In particular, this model concentrates on the bone/muscle functionality and the logic of their control whereas the circulation of information in the body of an HI is presented as if it was processed by a conventional computer. An HI can be considered a human-like robot with its memory organized like that of a conventional PC with the muscles being autonomic devices attached to the body in the same manner in which devices such as a CD-ROM are attached to a personal computer. On the other hand, if needed, the properties of an HI can be designed to match the characteristics of real human beings.

According to this model, the extended abilities of a human being for producing and using programs is its most distinct feature basically distinguishing it from any other creature.

3.2.3.1. Moving Parts

A moving part consists of a bone or an organ with an attached muscle. A unit includes an array of muscle threads and a command port accepting the values *contract* and *relax*. The command *contract* sent by the brain by the intermediary of motor neurons increases the contraction of the muscle to a certain degree and the command *relax* decreases contraction. The commands are transmitted from the brain to the muscles by the intermediary, here nerve fibers, which are viewed as types of wires.

3.2.3.2. Input Ports (Senses)

A sense consists of receptors hard-connected with input ports (volatile variables in C/C++). Change of a receptor automatically changes the state of the respective port. A group of receptor-port pairs could be viewed as an array of variables, whose states can be queried through their ports. Simultaneously, an HI receives information from millions of receptors organized in various arrays. For example, cutaneous receptors of a finger can be considered a system of 13 arrays, that is, three fingers' phalanxes; each equipped with four arrays placed on each side and the last array placed on the fingertip (it would be 9 arrays for the thumb). Cutaneous receptors are stimulated when a surface with receptors contacts an external object, that is, the array receives a value composed of the values of all array elements.

There are more senses than just the five traditional human senses of sight, hearing, taste, smell and touch. Additional senses include pain, equilibrium (balance), the aforementioned feelings of muscle contraction, the kinesthetic (motion) sense, the senses of temperature etc.

In contrast to the receptors of traditional senses affected by the forces of external environment, receptors of the additional ones are influenced internally. Thus, receptors of balance are a part of the vestibular apparatus; receptors of muscle states are internal variables allocated in the brain and changed during the process of muscle change.

3.2.3.3. Nervous System

The nervous system is a set of wires and activators connecting the brain with moving parts (output signal) and senses (input signal). Differing from conventional copper wires needing external energy for signal transmission, neurons transmit signals with the help of their own energy source.

A motor neuron is regarded as a combination of an activator and a wire filtering out input information.

3.2.3.4. Brain

The brain is considered to be the computer that controls the moving parts and receives information from the input ports (senses) much in the way as digital computers.

The internal memory of the brain contains *images*, which are sets of receptor states stored while executing programs.

3.2.3.5. Programs (routines)

A simple routine consists of a sequence of either operators, which are commands to muscles moving respective body parts or conditional operators making choices based on the changes of various receptors (feelings).

For example, making a fist is a set of actions including the contraction of the muscles of the four fingers and the thumb during which the brain is informed about the stages of muscle contraction and the physical contacts of fingers and the palm. Differing from classical computer programs, these are mainly parallel algorithms including simultaneous actions of several moving parts (all the fingers of a hand are clenched at the same time to make a fist).

Complex routines are skills such as walking, talking, riding etc.

The upper level is represented by activities, which are sequences of routines targeting a goal, like preparing food, going from point to another, studying a subject, etc. Activities can be extremely complex and take many years to achieve such as making a career.

The process of self-programming an HI starts immediately after birth and continues until death. This view also includes a detailed description of the self-programming algorithms as well as the ways in which these programs are used.

3.2.3.6. External Things and Internal Images

The execution of a routine is accompanied by feelings generated by the various senses. The input from the senses goes to the biocomputer, where it is routinely compared with the previous state. The differences between the actual and awaited changes are used to start, to change the command flow, to prolong and to stop an executed program.

Distinct executions of the same routine produce distinct feelings if there are differences in participating entities. Making a fist e.g. will produce distinct feelings because of injuries and diseases of the palm and the finger. The main distinction, however, is the difference in the external environment: The different taste feelings that will be produced by eating different food, the different feeling of touch that will be produced by walking on different surfaces, the different sense of smell that will be generated by smelling different things etc.

In contrast, the same external thing in the same state will produce similar feelings during execution of the same routine and these feelings are used by the HI’s brain as the indication of these things.

Normally the same external thing produces many various indications during the execution of miscellaneous programs, all of them creating the internal image representing this external thing. Images are collections of indications and muscle controlling commands allocated in the HI’s brain. For example, an image of an apple includes such indications as visual images, taste, touch, and smell perceptions. The indications are generated during certain actions. Thus, when an HI moves closer to an apple it gets a sequence of visual images, each containing a larger and more detailed picture of the apple. Accordingly, when an HI backs away (or the apple is moved away for some reason), the eyes produce a series of diminishing apple images. Also, the view is changed when the apple is seen from different angles revealing the part of the apple hidden in previous perspectives.

Other senses are involved when an HI comes into direct contact with the apple. Specific tactile sensations are received during the process of picking up the apple. The HI smells the specific smell when the apple is placed under its nose and a specific gustatory sensation when chewing the apple.

Images are the building blocks of the basic data representation system guaranteeing the effectiveness of an HI functionality. As soon as an HI sees an apple, the complete set of information associated with various algorithms such as moving towards, picking up and chewing, is activated by the brain, enabling the HI to make a decision, for example, whether to pick up the apple.

Images create the layer of the private information (tacit knowledge) that cannot be directly communicated to other persons as it is built on the base of perceptions and commands to various muscles. Since different HIs have distinctly different experiences, their images are different. A general way to get the same tacit knowledge for different HIs is to compel them to study the same set of external entities, for example, using the same books, learning the same theories.

Images are fairly large because they contain information produced by millions of receptors. Yet, the size of the data is reduced due to multiple compressing algorithms producing shrinking representations of images on the basis of miscellaneous heuristics.

An image type (the notion of a thing) is the generalization of images of similar things produced by an HI. Concrete images are (sets of) values of image types.

3.2.3.7. Changes (Procedures)

A change to an external thing is perceived by the HI's senses as the change of the value of the respective image type. It is the only form of information that can be communicated between HIs. The prerequisite for successful communication is that similar images are allocated in the brains of communicators. When images are different, the only information that can be exchanged is that relating to the common parts of them. Let us assume there are two HIs – one color-blind and the other non-color-blind - obtaining information relating to the status of a vertical traffic light from a third person who is not color-blind. Both will understand statements like "the upper light is switched on", but the statement "the red light is switched on" would be understood differently because the HI suffering from color blindness is not able to differentiate between green and red. As a result, this HI will find such information unintelligible or interpret the expression "red light" as just another name for "upper light".

Changes to things are correlated with changes of their indications but these relations are not always direct. Thus, the visual image of a balloon perceived by an HI will grow when the balloon is inflated but also when it gets closer to the observer and/or the observer moves closer to the balloon. An HI is normally able to distinguish between these cases, because it makes its conclusion not on the basis of a single visual balloon image but also on the basis of other perceptions like recognizing the change in distance between a balloon and itself as well as the fact of its own movement.

The identification of correct change is extremely important as it is used by the brain to decide what to do in a certain situation. If an HI registers an increase in size of a certain image, for example, an image of a wolf, it understands that the distance to the wolf is shortened and so it can react by running away or preparing to fight.

3.2.3.8. Human Language

Indications of external entities are direct when an HI comes into direct contact with the external thing or indirect when an HI interprets other things as indications of external entities without whatever contact with the latter. For example, an HI sees a wolf (direct indication), or a trace of the latter (indirect indication). In its simplest form, a human language is nothing more than an alternative set of indirect indications. That is to say, the direct and indirect indications of a wolf are extended by the language indications produced by another HI, who indicated the presence of a wolf in some way and transmits this information with the help of a human language.

The main advantage of language indicators consists of the ability to use the knowledge of other HIs.

3.2.4. Software ICE

Even though the code of high-level programming languages often looks similar to the descriptions of the real world objects produced in a natural language, the semantics of this code is completely different from that of natural language. Unlike other ICE, software entities are not independent things and their real features cannot be understood separately from their physical implementations, which is normally not specified in the programming code. Software entities are programs executed by some hardware mechanisms whose characteristics constitute the decisional part of software semantics.

An important characteristic of hardware is its ability for reprogramming.

Non-reprogrammable hardware mechanisms execute unchangeable programs physically built-in into their body. To this subclass belongs such entities as unicellular microorganisms and multicellular organisms without conditional reflexes, Al-Jazari's programmable Automata as well as modern processors whose functionality is set by microprograms physically burnt into the processor's microchip.

The reprogrammable mechanisms, such as Jacquard loom, multicellular organisms with conditional reflexes, computers as well as other soft-programmed artificial devices, allow change of their programs hence enabling change of their behavior patterns.

Reprogrammable hardware objects are ***emulators,*** which can, depending on a program, emulate distinct logical ***emulations***.

However, the emulation abilities of natural ICEs are rather limited, because their reprogrammable features are smaller than those that cannot be changed. The ability to reprogram grows by the high developed species. The most remarkable reprogrammable ICE is a human being whose ability to reprogram themselves by acquiring new information fundamentally supersedes those of lower developed species.

In contrast to biocomputers, digital computers are ideal emulators that can be completely reprogrammed. A digital computer creates an emulation by running a program.

All emulators are ICEs normally possessing a large number of variers but that is generally not true for emulations. Only emulations with conditional code possess ICE functionality, those without conditions are not ICEs!

Expressional restrictions of a programming language can be briefly explained with the help of the example function `fz` implementing the algorithm of the assembler command JZ (Jump if Zero). The code of this function defines the pseudo-dynamic emulation mechanism including variables `vari`, `adr` and the activating switch defined by of the control structure `if`

```
int jz(unsigned int vari,void* adr)
{
    extern void * IP;
    if( !vari ) IP = adr;
}
```

Let us assume such a command is hardware implemented and it is necessary to describe the electronic circuit doing it. Even if the code of `jz` code provides the algorithm for switching, it completely fails to describe its other essential parameters like the strength and voltage of the electrical current, its physical dimensions, materials, physical principles etc. The conventional way of representing real ICEs consists of documenting their characteristics with the help of narrative texts, graphics, computer images and physical models if needed.

In practice, the pseudo-code descriptions using notation of a high-level programming language is often used for the representation of algorithms — the approach often followed in the documentation of low level assemblers. Such representations are able to describe certain characteristics of real objects while completely ignoring their real structure and functionality.

Because software entities are the only kind of objects that can be routinely represented with formal languages, they are used in many examples throughout this work. This does not mean that this work is restricted to the consideration of software ICE but only that there currently are no expression tools for expressing non-executable functionality. Actually, such functionality can be adequately presented in T but this language will be defined in the concluding part of this work and for this reason cannot be used during the depiction of the preceding material.

3.2.5. Knowledge

1. Historically, the first definition of knowledge was introduced by Plato in his "Dialogues" where it was defined as "justified, true belief" (Chappell 2013). In spite of a seeming pluralism, all currently used definitions are nothing other than numerous variants of this basic definition, differing by rather minor aspects such as natures of justification procedures, kinds of beliefs (explicit knowledge vs. tacit knowledge, knowing how vs. knowing that), relations between beliefs and real objects. Taking into account that thus far no knowledge model has proven to be really effective, we have come to the unavoidable conclusion that the central idea of Plato's definition – interpreting knowledge as beliefs – is fundamentally faulty.

The problem of this definition consists of the nature of beliefs, which by their definition, are person's mind settings defining how he/she thinks and handles. Just see following the definition of the term ***belief*** from Merriam-Webster ("Definition of Belief" n.d.):

- a state or habit of mind in which trust or confidence is placed in some person or thing;
- something believed ; esp: a tenet or body of tenets held by a group;
- conviction of the truth of some statement or the reality of some being or phenomenon esp. when based on examination of evidence.

While most complex types of knowledge are actually justified, true beliefs, the simple know-how do not require high-level brain activity and consist of inborn reflexes. A healthy newborn baby has absolutely no beliefs but still possesses the inherent knowledge how to suck a mother's breast.

TMI repairs this fault by replacing the obviously overstrained notion of belief with the notion of information. Knowledge, in TMI, is understood as justified, true information or yet better as *adequate information*. Adequate information is that allowing the reliable functioning of ICE. This concept can be well demonstrated on the example of a unicellular organism 3.1.4 changing its metabolism according to change of water temperature.

Knowledge of these microorganisms consists of ability to differ between two distinct environment states, each associated with a particular behavior pattern. This knowledge remains intact till the microorganism remains in its lagoon where the continued rounds of events occur according to the warmer-colder formula. As soon as such an organism moves out of this lagoon for any reason, it will not be able to survive because its knowledge become irrelevant.

However, if streams moving microorganisms away become regular, the process of natural selection will produce microorganisms adapted to the environmental changes, like those with ability to register two distinct environments, each with two distinct behavior patterns. Advanced microorganisms will get additional survival chances because of their superior knowledge system differing between four environment states.

There are two conclusions, which can be made on the base of the aforementioned example:

- Knowledge is an advantageous feature increasing the survival chances of organisms and organism groups;
- The only general way of proving the trustworthiness of knowledge consists in the additional study of the represented things and the owners of representations, but ***not*** in the study of the knowledge structures. Hence rather insufficient effectiveness of the methods of formal logic, which are traditionally ranking among the most powerful tools of knowledge representation. In reality, these methods can only be used for small number of second-rate tasks like elimination of contradictions in complex knowledge systems.

2. Information produced by sensors is essentially imprinted on an ICE by its environment. The totality of external objects producing their imprints (knowledge system) constitutes the ICE's ***known world***. Generally, not only human beings, but also any object of any structure and origin could be taken for the center of the knowledge system reflecting all entities with which this ICE can contact in this or other way. Thus, the known world of microorganism 3.1.4 consists of a water system with the aforementioned cold-warm pattern.

Distinctly to the inorganic ICE and living organisms of lower levels, human beings possess an open knowledge system. The feature of openness is introduced by the ability for indirect contacts between human beings and entities being out of the reach of the former and deep classification of these entities.

3. Systems with the ability to information processing, possess both the built-in and the acquirable knowledge systems.

Most of knowledge in the low-level living organisms is built into them from their birth. Consider the example, when a worker bee finds a source of nectar and returns to the beehive where it performs a complex dance routine to communicate to the other bees the location of this nectar. Depending on the type of dance (round dance for nearby and tailwagging dance, with variable tempo, for further away and how far), the other bees can work out where this newly discovered feast can be found (Yule 2006).

In this case, the complete information about the dance, all its variation and the purpose is the part of the built-in knowledge. The only acquired knowledge consists of the nectar location transmitted by the dance process.

Human beings have opposite relations between built-in and acquired knowledge. For example, a person, sitting on a balcony, is attacked by an annoying fly. Differently from a bee, a person can react in a number of ways. The most common variants are 1) instinctively try chasing this flay away; 2) ignore it,3) try to get rid of it by killing or chasing out; 4) leave the balcony.

There is a lot of acquired information and algorithms involved in the process of making and implementing decision:

- The identification of the attacker, which includes the input of external senses and the database enumerating all possible external objects. The identification process will find the matching object and finish.
- The identification of possible reactions. The output of the first step with addition of information about the person's position and the state of organism (could it move in the moment and how quickly) is used in this step to produce the aforementioned set of possible actions.
- Choosing most appropriate reaction what involves miscellaneous information from memory or/and external senses. For example, the variant (3) (kill or drive out) is chosen.
- The person counterattacks the fly what occur with consistent information flow from the external senses about the current fly location as well as the position of the person itself.

Regarding, the built-in knowledge, it is mainly present as the part of low-level algorithms of movements of the person and its body parts.

4. The Intelligence Theory

According to the definition in Wikipedia (“Artificial Intelligence” 2018) “Artificial intelligence (AI, also machine intelligence, MI) is intelligence displayed by machines, in contrast with the natural intelligence (NI) displayed by humans and other animals. In computer science, AI research is defined as the study of "intelligent agents": any device that perceives its environment and takes actions that maximize its chance of success at some goal. Colloquially, the term "artificial intelligence" is applied when a machine mimics "cognitive" functions that humans associate with other human minds, such as "learning" and "problem solving".

Though normally this online encyclopedia is not considered as scientifically reliable it is exactly the right choice, in this particular case, because one can await that the site getting circa 10000 clicks everyday probably contains the pretty adequate definition of its subject. The definition made on basis of (Poole, Mackworth, and Goebel 1998, 17; Russell and Norvig 2010; Nilsson 1998; Legg and Hutter 2007), demonstrates two key specifics of the modern AI research; first, AI is understood as the antipode of NI, which is the only form of the true intelligence known at this time; second, the task of explaining the phenomenon of intelligence is completely ignored because the focus of the research is concentrated on the study of the artificial intellectual entities no matter how “intelligent” they in reality are.

The Achilles' heel of this approach is that the feature of being intelligent is not a primary characteristic of someone (something) but rather an individual assessment of some observer judging the behavior of a watched thing. Attributing a certain thing with intelligence is exactly the same as attributing it with lightness, softness, bigness or goodness. In its genuine form, it is just an individual opinion assessing some particular feature, like “lightness”: - a thing which light for one person can be very heavy to another.

Exactly this characteristic of AI is very well demonstrated by the famous Turing test As write (Russell and Norvig 2010) “Alan Turing, in his famous paper "Computing Machinery and Intelligence" (1950) suggested that instead of asking whether machines can think, we should ask whether machines can pass a behavioral intelligence test, which has come to be called the Turing Test. The test is for a program to have a conversation (via online typed messages) with an interrogator for five minutes. The interrogator then has to guess if the conversation is with a program or a person; the program passes the test if it fools the interrogator 30% of the time. Turing conjectured that, by the year 2000, a computer … could be programmed well enough to pass the test. He was wrong—programs have yet to fool a sophisticated judge.

On the other hand, many people have been fooled when they didn't know they might be chaffing with a computer.”

The subjectiveness of a notion however, does not excludes its objective definition, which can be done if the internal characteristics of this notion obtain explicit characteristics. Thus, by agreeing that the weight lesser than 10 kilos is light and that bigger than 80 kilos is heavy, makes notions of lightness and heaviness objective.

In case of intelligence, this require the developing of measurable and unambiguous specification of this feature – the task never actually perceived by AI community. Instead, the modern AI research is fully concentrates on the intellectual features and so makes further mistake by disregarding the differences existing between intellectual tools and their masters. All currently known masters are human beings, only they possess true intelligence, while a tool is basically not obliged to possess any intellectual abilities. But also if a tool has any such abilities, they are still not sufficient to make a task perceiving by a master who is able to use different tools for implementing a particular task. Moreover, a master functioning as a tool does its task in completely different way as a pure tool and in most cases, the latter simply emulates the master's functionality. It is clear that intelligence of tools has to be studied separately from the intelligence of masters, but currently such study is impossible because all currently studied intelligent agents are specialized tools with essentially restricted intellectual abilities.

Another weak point of the modern AI research is its concentration on neuron nets. Despite they are very popular just now, their general value is obviously overestimated. The fact that NI uses a neuron brain does not mean that the neuron-based organization has no alternative in the AI domain. To the contrary, at the beginning of AI, neuron nets were seen as the second best solution. Implementations based on conventional digital computers were always preferred. In view of the modern understanding of a brain, it could well be seen as a specific implementation of the intellectual functionality, which is still remains the most efficient implementation but the persistent growing number of working systems developed on conventional computers suggests that the latter direction obviously has greater potential. Furthermore, neuron nets are not safe, none knows how they come to their conclusions and it is very dangerous to leave the intelligent activities to the black boxes.

The intelligence theory considered in this chapter goes completely different way based on the necessary and sufficient definition of intelligence, which remains actual for both NI and AI. Its main points are detailed in the Section 4.2.
The following section exposes the theoretical nuisances of AI research.

4.1. The Failure of AI Theories

4.1.1. The Optimistic Beginning

Although the philosophical history of AI can be traced back hundreds of years, this modern discipline has its roots in post-World War II development and research when technological development was sufficiently advanced so scientists could start probing into creating an artificial brain or thinking machines. The neurological research has already shown that the brain has an electrical network of neurons that fire in all-or-nothing pulses. Walter Pitts and Warren McCulloch analyzed networks of idealized artificial neurons and showed how they might perform simple logical functions. They were the first to describe what later researchers would call a neural network. One student inspired by their work was 24-year-old Marvin Minsky who together with Dean Edmonds built the first neural net machine SNARC in 1951. Claude Shannon developed his information theory in 1948 and Norbert Wiener created the new field of cybernetics, describing control and stability in electrical networks. Equally important, Alan Turing showed using his theory of computation that any form of computation could be described digitally. These developments were the impetus for discussions regarding the possibility of constructing an electronic brain. (McCorduck 2004; Crevier 1993; Russell and Norvig 2010). All these ideas enabled the discussion about the possibility of constructing an electronic brain.

The more practical applied area of games closed in on the same field from a different angle. In 1951, with the help of the Ferranti Mark 1 machine at the University of Manchester, a checkers and chess program were written by Christopher Strachey and Dietrich Prinz, respectively (Copeland 2000). Arthur Samuel, working on the problem from the 1950s to 1970, finally developed a more advanced checkers program that was able to match amateur players (Schaeffer 2009, Chap. 6). Game AI thus became and continues to be an indicator of how far AI has come as a field, up to the point that today it is often the common language use of the term AI.

Algorithms were developed to solve increasingly complex mathematical problems. In 1955, Allen Newell and (future Nobel Laureate) Herbert A. Simon created the "Logic Theorist" - a program that went on to prove most theorems of Russell and Whitehead's Principia Mathematica, including hitherto unknown ones. According to Simon, this was seen as a solution to "the venerable mind-body program, explaining how a system composed of matter can have the properties of mind (cited in: Russell and Norvig 1995, 17; Crevier 1993).

AI saw its evolution as an academic discipline at the Dartmouth Conference organized by Marvin Minsky, John McCarthy, Claude Shannon and Nathan Rochester in 1956. It was then that the name AI was commonly accepted for the field of "the science and engineering of making intelligent machines" as John McCarthy defined it (McCarthy 2007). The conference proposal stated in its core idea that "every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it" (McCorduck 2004, 111).

This rapid development produced great optimism. In 1965, Simon assumed that "machines will be capable, within twenty years, of doing any work a man can do." (Simon 1965, 96) In 1967, Marvin Minsky stated that "Within a generation ... the problem of creating 'artificial intelligence' will substantially be solved." Three years later, he put it even more succinctly in Life Magazine: "In from three to eight years we will have a machine with the general intelligence of an average human being." (McCorduck 2004, 272–74).

Obviously, these predictions turned out to be far too optimistic.

4.1.2. The Current State of AI Science

Contemporary AI is studied by an extensive multidisciplinary field including such disciplines as computer science, mathematics, linguistics, psychology, philosophy and neuroscience. The optimistic predictions have still not been realized, but the engineering part of AI appears to have been pretty successful. Language translation, intelligent search algorithms, chess programs, robots performing complex surgical operations, automated car driving systems and other applications have resulted in remarkable achievements. The software development of such applications does not differ from the programming of complex "non-intelligent" applications, as both areas require building millions of lines of code (mainly in C/C++) running on complex hardware systems.

The success of AI engineering in contrast with the fundamental AI theory is even more startling. Theoretical assumptions are hardly used in engineering projects and their predictive power (which is the essential part of any relevant natural science) is close to zero.

This is clearly demonstrated in recent discussions regarding the potential danger of AI. The discussion's backbone was formed by commonsense considerations based on everyday experiences, completely lacking any theoretically sound arguments. The characteristic excerpts are referred to below. Due to the fact that there are virtually no such discussions in academic fora, the excerpts are from journalistic outlets.

Elon Musk in Washington Post: "With artificial intelligence we are summoning the demon. In all those stories where there's the guy with the pentagram and the holy water, it's like yeah he's sure he can control the demon. Didn't work out." (Moyer 2014).

The prominent physicist-theoretician Stephen Hawkins: "It (AI) would take off on its own and re-design itself at an ever increasing rate. Humans, who are limited by slow biological evolution, couldn't compete, and would be superseded." (Cellan-Jones 2014).

The prominent AI proponent Ray Kurzweil quoted by Time, appeals to recent history, completely avoiding any theoretically based arguments: "If AI becomes an existential threat, it won't be the first one. Humanity was introduced to existential risk when I was a child sitting under my desk during the civil-defense drills of the 1950s. Since then we have encountered comparable specters, like the possibility of a bioterrorist creating a new virus for which humankind has no defense. Technology has always been a double-edged sword, since fire kept us warm but also burned down our villages." (Kurzweil 2014; Luckerson 2014).

The miserable state of AI science, which is not even able to provide any consistent arguments in such an important discussion, is a consequence of its multidisciplinarity. The latter could be advantageous in learning processes, thereby enabling the same objects to be studied from different viewpoints, however, it is not a working instrument. A science matures by developing unifying theoretical constructs and not otherwise.

Physics is a case in point. Until rather recently, the terms physics and natural philosophy were used interchangeably for a science whose aim was the discovery and formulation of the fundamental laws of nature. Present day physics defined as the study of matter, energy and the relation between them, was developed by finding common laws uniting previously unconnected disciplines as optics, mechanics, magnetism, thermodynamics, and so on.

The fact that no AI-related discipline was ever able to establish its dominance by presenting a unifying theory, could be considered as an indication of the **perspective-boundedness** of all these disciplines. AI from a linguistics viewpoint is concerned with human language alone, the approach of epistemology only with human knowledge, the study of decision-making exclusively with the process of making decisions; but none of them has ever been able to propose a perspective independent approach, which could be used to explain all AI relevant characteristics.

Taking into account that almost sixty years have passed since the Dartmouth Conference, one would believe that this has been more than enough time to formulate any unifying theoretical constructs if the embryo was ever present in one of AI-related disciplines. Apparently, it was not the case.

4.1.3. The de-facto AI Paradigm and its Conflict with Reality

The perspective-boundedness of AI disciplines resulted from the initial understanding of intelligence in the 40s and 50s. These were the constituting years of AI, in which software development was taking its first steps. There were no working AI algorithms and it was unclear how to create them. So it is not surprising that human-related intelligent activities happened to be the exclusive source lacking methods and theories.

Once collected and studied, these activities predefined the dominating thought pattern (paradigm) of AI, according to which **only humans can really think** while machines can only emulate human thinking by executing the respective algorithms. Hence, emphasis on such research directions like building AI functionality on the basis of neural nets, expressing the thinking process on the basis of mental and cognitive states and creating thinking machines by writing algorithms based on logic and mathematics.

This paradigm actually predefined the following AI development, however not because of its truthiness (in reality it was wrong) but because the scientific community utterly concerned with particular tasks failed to understand the fundamental constraints of its own approach. Formed as a result of random practical activities, this paradigm occurred to be implicit and because no theory can operate with implicit concepts, the scientific community effectively restricted its own research by making equation between the human thinking and the thinking per se. This was well seen in the lively discussion on the topic started by Alan Turing long before the Dartmouth Conference.

Initially AI scholars were divided in two camps[6] based on the question whether machines could really think, and if they could, whether thinking machines must be modeled after humans or using abstract models. As the years passed by and the enthusiastic predictions of the more optimistic members of the discipline did not materialize, the opponents of "intelligent machines" who saw an inherent contradiction in the term "Artificial Intelligence" prevailed. The real problem, however, was not the assumingly contradictive term, but the multidisciplinarity. The AI field obviously turned out to be too disperse, the methods of its sciences too distinct and too specialized for allowing the fundamental interdisciplinary discussions.

In retrospect, it is clear that the human-centric approach was on the right path during the early AI years. Problems arose later when computers started to solve previously unsolvable intellectual tasks in their own way using their plentiful hardware resources. This would have been the right time to change the prevailing approach by considering the previously unknown methods as acts of true thinking, but instead the scientific community simply ignored the latter, causing the gap between practice and theory. The history of computer chess is a classic example of this development.

In his work Claude Shannon (Claude E. Shannon 1950), designed the basic outline of a chess program and as soon as in 1957, Herbert Simon predicted that in ten years, a computer would defeat a living, breathing chess master. They based their predictions on

[6] Russell and Norvig identify four perspectives, crossing the two dimensions of behavior vs. thinking and human vs. rational (Russell and Norvig 1995, 1–2).

refutation screening, i.e. the application of alpha-beta pruning for optimizing the evaluation of a move. Missing from their computations, however, was the ability to determine the right order when evaluating branches. One method focused on killer heuristics, i.e. unusually high scoring moves to reexamine when evaluating other branches. Despite all efforts, most top chess players were deeply incredulous of claims that computers would soon be able to play at an expert level even a decade after the predicted date. (cf. McCorduck 2004)

As Fred Hapgood described in the New Scientist in 1982:

‟As late as 1976, most serious chess players thought that the day when computers would play master-level chess was still far off. In that year a Senior Master and professor of psychology Eliot Hearst of Indiana University wrote that “*the only way a current computer program could ever win a single game against a master player would be for the master, perhaps in a drunken stupor while playing 50 games simultaneously, to commit some once-in-a-year blunder…*” (Hapgood 1982, 829)

As often happens with bold predictions, it was proven wrong that very year. For the first time, a computer won a tournament: a program developed by Northwestern University came in first in the Class B section of the Paul Masson American Chess Championship. Hapgood goes on to give us a glimpse at the progress and thinking of that time:

“The winning streak caught a lot of chess players by surprise. It was hard to see a good reason for it. There have been no conceptual breakthroughs in the science. The machines were still unable to plan: their routines for evaluating moves were as crude as ever. The only real progress has come in learning how to drive those evaluation routines faster and faster, thus allowing larger numbers of position to be examined in the time available […] Programmers Dan and Kathe Spracklen, who have paced the industry since their Sargon program won the first micro chess tournament in 1978, believe that 90 per cent of this progress is due to faster speeds in the evaluation routines, and only 10 per cent because the routines themselves have become more chess-intelligent.” (Hapgood 1982, 829–30)

As can be seen from the description above, the sudden surge in playing power was a big surprise because it did not follow then dominated theoretical assumptions of computer chess. The surge downgraded the previously scientific problem to the pure engineering solution but made no contribution to the theoretical constructs, instead the theoretically based approaches were put on the back burner. Later the same way will be followed by the natural language processing, image processing and further intellectual tasks.

A non-working science existing in parallel with the purely pragmatic development is not the sole prerogative of AI. The same situation once existed in programming (see Chapter 2 and Section 5.4 of this book).

In retrospect, it is clear that both the programming language theory as well as the AI field suffered the same central problem, which is the **initially incorrect conceptualizations never corrected afterwards**. Furthermore, both disciplines committed the same mistake by viewing non-mathematical entities in mathematical terms. That this might be detrimental to the further development of the field was already recognized by one of the pioneers of AI, Herbert Simon. Asked why the early rapid development of AI did not last for a longer time, he pointed to the lack of revolutionary fervor among researchers and added that "we also underestimated the extent to which the computer-science culture was going to be colored by the mathematics culture during the early years, and heuristics never appealed to mathematicians – there weren't any theorems in it!" (McCorduck 2004, 222)

Currently, AI engineering is moving away from AI science in exactly the same way practical programming did from its rather irrelevant theoretical basis. However, AI is not programming! AI applications have the potential to act on their own. Thus, the question that comes to mind is – what will happen if engineers finally find the necessary solutions without the assistance of AI science? Will pragmatic-driven developers retain control of their creations? Or will we see a realization of the dark scenarios described by Elon Musk and Stephen Hawkins?

Certain negative results produced by the programming language theory are highly topical for the AI field and are considered in the following two subsections.

4.1.4. Turing machine

The concept of the Turing machine was very helpful in building the mathematical abstraction of computers, but turned out to be completely irrelevant in programming and extremely harmful to AI.

In programming, this concept was used by the programming language theory for defining the criteria of language universality. During the three first decades of high-level programming, this was one of the most important issues, because the contest for the best programming language was mainly the context for the most universal programming language. Bad luck for the theory whose considerations turned out to be completely useless for practical developers. The reason for this was the concept of the Turing machine, according to which all programming languages are equally universal. Actually, it was the second factor after Moore's Law, which contributed to the demise of the programming language theory.

The concept of the Turing machine harmed AI in two distinct ways:

1) By overestimating the issue of computability (executability) of an algorithm. In the real world, concrete implementation of some algorithm is secondary relative to its role in

the enclosing system. If one wants bread, one can get it in different ways like getting it from the breadbox in the kitchen, buying it at the bakery, making it from scratch and so on. In programming, this feature is expressed by the concept of the function's prototype, which can be associated with distinct implementations. Prioritizing the algorithm's computability before the algorithm's role leads to a concentration on the particular implementations of intellectual functionality. The characteristic example is the hugely exaggerated issue of neuron nets, which are often viewed as the Holy Grail of intellectual functionality.

2) By using the non-utilizable understanding of computability. The Turing machine is ineffective and non-intuitive to such a large extent that it is simply impossible to use (or its concept) for anything having a practical purpose. In fact, this concept discredits a theory in the eyes of practical developers – the same development, which once occurred in programming.

4.1.5. Epistemology

Plato's definition of knowledge as "justified, true belief" is relevant for the characterization of the most complex kinds of knowledge but fails in the case of simple know-how, which does not require high-level brain activity and consists of inborn reflexes. See more in the Section 3.2.5.

4.1.6. Formality

The issue of formality is one of the most important in AI. The issue was raised by Turing who made the "argument from informality of behavior" The Stanford Encyclopedia of Philosophy characterized it in the following way: "This argument relies on the assumption that there is no set of rules that describes what a person ought to do in every possible set of circumstances, and on the further assumption that there is a set of rules that describes what a machine will do in every possible set of circumstances. From these two assumptions, it is supposed to follow—somehow!—that people are not machines." (Oppy and Dowe 2011)

According to Russell and Norvig "Essentially, this is the claim that human behavior is far too complex to be captured by any simple set of rules and that because computers can do no more than follow a set of rules, they cannot generate behavior as intelligent as that of humans. The inability to capture everything in a set of logical rules is called the qualification problem in AI." (Russell and Norvig 1995, 1024)

Searle used this notion in his axioms for producing opposition between programs and minds:

Computer programs are formal (syntactic).

Human minds have mental contents (semantics).

Syntax by itself is neither constitutive of nor sufficient for semantics.
Conclusion 1: "Programs are neither constitutive of nor sufficient for minds."
Conclusion 2: "Any other system capable of causing minds would have to have causal powers at least equivalent to those of brains. (Searle 1990).

The above citations are quite representative in characterizing the understanding of formality in AI. In short: minds are informal and have mental content, while machines act on the basis of formal rules which prevents them from executing tasks carried out by minds.

This opposition however allows for a completely different explanation, which has nothing to do with the supposed differences existing between real minds and real machines. Confronted are **two different worldviews** – the one applied to known objects while other considers objects whose features are not completely known. A computer is a device of the first type, also if a person has no knowledge of the running programming code it knows that there is one and that all functionality of a computer is controlled by its code. It is different in the case of a human, because at this time there is no exact knowledge regarding the inner workings of a person's head. In other words, in the eyes of an external observer, a human being is seen as a system with unknown unknowns, which makes the explicit description of a human mind impossible.

In general, the qualification problem can also be detected by conventional software when a formal program acts in such a way that an external observer sees it as behaving informally. Any experienced programmer working with complex software periodically encounters such things in the form of volatile bugs occurring without recognizable patterns for unclear reasons. Like with any other bug, such a bug can be localized and fixed but usually only after a time-consuming and annoying debugging process. After reproducing the exact program state causing such a bug, it will lose its informality and will be seen as yet another conventional mistake occurring due to some particular combination of distinct data elements.

The informal activity of a human occurs for the same reasons and in the same way. An observer sees a person behaving informally because it does not have the complete information regarding the reasons causing the person's behavior. The missing information can be tacit knowledge of the observed person; it can also be the particular recollection of something and so on. The brain of a conscious person works continuously and the complete information used to make a particular decision does not include just the description of the subject of this decision but also the totality of the actual brain and mental states.

Taking into account that a brain of a real person contains a very voluminous database formed over the period of its whole life, the decisions it can make on the same issue at different times can differ unpredictably. A machine behaves exactly the same way if programmed and supplied with the same data. As Herbert Simon once put "Human beings, viewed as behaving systems, are quite simple. The apparent complexity of our behavior over time is largely a reflection of the complexity of the environment in which we find ourselves"(H.A. Simon 1969, 110).

4.1.7. Searle's "Chinese Room": Why this thought experiment is wrong.

The problem of understanding is one of the paramount concerns in the AI field. John Searle formulated the critical position regarding the ability of a computer to understand in his thought experiment – the Chinese room. The intent of this experiment was to disprove the equivalence of human and computational intelligence.

The setting is described by the Stanford Encyclopedia of Philosophy as follows: "Searle imagines himself alone in a room following a computer program for responding to Chinese characters slipped under the door. Searle understands nothing of Chinese, and yet, by following the program for manipulating symbols and numerals just as a computer does, he produces appropriate strings of Chinese characters that fool those outside into thinking there is a Chinese speaker in the room." (Cole 2014).

Furthermore, Searle argues that if there was a computer program that allowed a computer to carry on an intelligent conversation in a written language, the computer executing the program would not understand the conversation either. Hence - minds are not machines. (Searle 1990).

Searle's argument actually confuses one's understanding with the externally visible proof thereof registered by some observer. These are however two completely different things.

Understanding itself is a purely internal process, which may in some cases produce externally visible signs while in others not. The conclusion of an observer about the existence of something or occurrence of some event made on the basis of externally visible signs or logical analysis is not equivalent to the actual existence/occurrence of this thing/event. A genuine understanding of something is the process of associating a certain predefined sign with a complete set of associated algorithms. The only reliable proof that one really understands something is to check whether one really possesses the necessary associations.

This is also not just exclusive to humans. Animals as well as computers can also understand. Also a Visual Studio editor possesses such abilities. Typing the string like "`myInstance.`" in the editor's window causes the latter to respond with a displayed list of members and methods of the class `myClass`, if `myInstance` was previously defined as an instance of the class `myClass`.

Back to Searle's argument, it is exclusively about proof, while the genuine process of understanding remains outside the context of discussion. If one listens or reads Chinese and answers in the same language, we may conclude that one understands this language. Such a conclusion can be wrong however in several borderline cases. For example, consider a person with a very good memory who can memorize a Chinese phrase book or parts thereof and who will be later questioned about what he memorizes. Another example: instead of making the above-described manipulations, Searle quietly transmits the input text to someone with Chinese understanding and the latter answers in the opposite direction.

Furthermore, consider the opposite case when a person in the room can really understand because this person is familiar with both English and Chinese. Generally-speaking we can conclude that the room's inhabitant can understand Chinese if he translates the text, but we can say nothing to the contrary because his failure to understand text on the unknown theme is not equivalent to the missing ability to understand the Chinese in principle.

To get the reliable proof of understanding, a questioner must thoroughly test the person sitting or standing before him, otherwise there is always the possibility of deception. The fact that an examined object is human is also a part of the proofing criteria because in the case of a non-human responder such a test would be irrelevant. Assume a questioner examines a computer, and the latter connects the questioner with a true human responder by means of Wi-Fi, radio waves, etc.

The above considerations allow for the following conclusions:

1. The understanding of a black box is not comprehendible in principle. This is exactly the reason why an examinee taking an exam cannot use mobile devices, has to prepare answers in a controlled room, answer verbally before an examiner and so on. If any possibility of cheating remains, there is always the danger of just another Chinese Room.
2. A computer remains a black box also when it is completely isolated from the outside environment. Assume a computer successfully answered all questions; it can still emulate understanding by querying the database containing all questions and answers ever made in Chinese. Google and other search engines use exactly this method, which however has nothing to do with the real understanding of a human language.

3. The only reliable way to convert a computer to a white box consists of making an algorithmic definition of understanding like the one used in this article. This however means that humans are machines, making Searle's argument irrelevant.

4.1.8. Brain in a Vat

In the field of AI, the problem of knowledge reliability is traditionally linked with the trustworthiness of beliefs. An example of this approach is the well-known thought experiment brain in a vat.

"Brain in a Vat is an element used in a variety of thought experiments intended to draw out certain features of our ideas of knowledge, reality, truth, mind and meaning, it is based on an idea, common to many sci-fi stories, that a mad scientist, machine, or other entity might remove a person's brain from the body, suspend it in a vat of life-sustaining liquid, and connect its neurons by wires to a supercomputer, which would provide it with electrical impulses identical to those the brain normally receives, according to such stories, the computer would then be simulating reality and the person with the 'disembodied' brain would continue to have perfectly conscious experiences without these being related to objects or events in the real world. […]

Since the brain in a vat gives and receives exactly the same impulses as it would if it were in a skull, and since these are its only way of interacting with its environment, then it is not possible to tell, from the perspective of the brain, whether it is in a skull or a vat, yet in the first case most of the person's beliefs may be true, such as if they believe, or theoretically say, that they are walking down the street, or eating ice-cream; in the latter case their beliefs are false" (Kennard, n.d., 126).

According to the approach of TMI, this thought experiment is concerned exclusively with the specifics of experienced knowledge, which differs from communicated knowledge, in that it complies with no general standards. This knowledge consists of images, which in most cases actually coincide with external things. Regarding cases without such a coincidence, they have absolutely nothing to do with the truthfulness of beliefs in the sense of mathematic logic, but rather with erroneous perceptions (mirages), forgetfulness and imagination.

An illiterate person, who was born and grew up in a desert, possesses a completely different view of the world as that of an illiterate person living on an ocean island, not to mention a forest inhabitant's view of the world. There are also other very specific habitats like mountains, steppes and even caves, where the inhabitants of these possess absolutely incompatible systems of experienced knowledge.

In this view, the brain in a vat is just the brain of a person living in a very specific environment – namely in a computer. It depends exclusively on the mad scientist qualification whether it possesses enough abilities for composing programs emulating the consistent environment. If yes, this brain will function in exactly the same way as do the brains of all other persons living in specific environments, otherwise it will collapse under a stream of non-organized impulses.

Experienced knowledge is the genuine reprint of the person's environment. If a person sees a chair, it recalls the set of associated algorithms meaning that a chair can be used for sitting, can be moved when not fastened, and so on. Brain in a vat has to react in the same way when getting the image associated with feelings of sitting. In case there are no chairs in the computer's world, it will produce whatever other images, which probably have no analogues in our world.

This is no different in the aforementioned specific worlds. A person living on an island cannot have the same notion of a sea as a person living in a closed cave system. The latter will also miss numerous other important things including the sun.

Experienced knowledge is complemented by communicated knowledge, which consists of references to the experienced knowledge. Knowledge which cannot be associated with respective internal images cannot be communicated in principle. The main advantage of communicated knowledge is its ability to expand. Once generated, an idea can be changed, extended, purified by many different persons with very different individual experiences. After heading in this direction, communicated knowledge became standardized. All abstract notions, including the concept of belief, are defined in the terms of this knowledge and has absolutely no sense in the basic world of experienced knowledge.

Persons possessing non-standard experienced knowledge are able to standardize their knowledge systems by communicating with other people in verbal or written form. Another point is that each of them has their own way of accomplishing that. Thus, a desert inhabitant can get a notion of the sea by comparing the unbounded desert's sand often making waves, while an inhabitant of a mountain can understand the comparison with the sky and clouds.

4.2. The TMI View of Intelligence

The intelligence theory is based on the model of an HI described in Section 3.2.3. This model complies completely with the requirements of a strong AI formulated by Joseph Weizenbaum. According to him, to create a strong AI is "nothing less than to build a machine on the model of man, a robot that is to have its childhood, to learn language as a child does, to gain its knowledge of the world by sensing the world through its own organs, and ultimately to contemplate the whole domain of human thought". (Weizenbaum 1976).

An HI is understood as a machine, whose organization and complexity allow the complete specter of intellectual functionality as providing the self-sustained existence with the help of knowledge acquisition, learning, adapting to the changed condition as well as goal setting. Since the intellectual activities of an HI are nothing other than sophisticated information processing, a TMI can be well considered as a consistent intelligence theory allowing explicit expression of all intelligence phenomena.

The approach of TMI is fundamentally different from all theories currently existing in the AI area, in particular because it rejects any explanations of intelligence made in the categories of mathematics and formal logic. This is a true white box approach, which allows explicit expression of all relevant algorithms also those currently implemented with the help of black boxes like neuron nets. Its model of HI allows composing the adequate specification of intelligent algorithms and so provides a thorough understanding of intellectual functionality.

4.2.1. The Intelligence Paradigm

The proposed theory is based on the assumption that **machines can really think.** This complies with Searle's view of a strong AI - "The computer is not merely a tool in the study of the mind, rather the appropriately programmed computer really is a mind in the sense that computers given the right programs can be literally said to understand and have other cognitive states" (Searle 1980, 417). The intelligence paradigm based on this assumption provides a completely different insight into the problem in relation with the traditional approach, particularly because it rejects the view of AI as a phenomenon while favoring the view based in definable und reproducible features.

Following this paradigm, there are two different types of intellectual systems: a natural intellect (NI), whose higher form is a human but which also includes the species on the lower branches of biological evolution, and the artificially created intelligent systems implemented by modern computers, robots and other computerized devices. They are designated here as DI (digital intelligence) because currently all these objects are powered by digital processors. NIs and DIs have very different intellectual abilities, but it is logical to assume that these differences "will eventually go away by itself once machines reach a certain level of sophistication" (Russell and Norvig 1995, 1027).

Despite the fact there are no real contradictions between NI and DI, both types of intelligence possess a lot of perceived inconsistencies originating from distinct conceptualizations of these entities. The NI study involves miscellaneous traditional conceptualizations of the intellectual activities of a human with the help of formal logic, mathematics, language understanding, epistemology, etc. Another conceptualization source consists in the dissection of a human's brain logically or physically (neuron nets).

The study of DI consists mainly of studying different specifications (the software code) of the intelligent functionality.

The idea of dissecting a human's brain in order to understand a human's intellect is not much different from the idea of dissecting a laptop in order to understand the functionality of Microsoft Windows. These complex systems cannot be understood with the help of dissection. They also cannot be understood with the conceptualizations of their externally visible activities as do formal logic etc. The latter are high-level phenomena based on completely different fundamental mechanisms.

The path to understanding intellectual algorithms lies in exposing the nature of intellectuality. This theory considers intelligence as a feature of sophisticated programmed systems and the difference between intelligent and plain software consists of complexity and sophistication. The intellectually relevant part of DI includes several multitier data processing systems. Thus, a modern laptop, which is quite sufficient for many DI applications, includes a physical processor running an operating system, which in turn, runs the required applications. The complete system has at least 14 levels of organization including:

1. At least 4 hardware levels of a processor organization including:
    - Electronic components like diodes, transistors, condensers
    - Logic gates implementing simple Boolean operations like AND, OR, NOT
    - Large integrated circuits, uniting thousands up to millions of elements of previous levels
    - Processor components like arithmetic logic unit (ALU), memory, control unit, memory management unit (MMU) etc.
2. Operating system, around 4-5 or more levels of software including drivers, physical and virtual access to the memory, the support of the process table etc.;
3. Software applications, roughly 6-8 levels of software implementing applications of the complexity comparable with that of Microsoft Word.

The growth of the intellectuality of software applications that has occurred in the last thirty years can be easily understood when comparing modern computing environments with their predecessors used thirty years ago. The differences between computing environments include the following factors:

1. *Computer performance*. At the beginning of the 1980s, mainframes did several million operations in a second and possessed several Mb of memory. The size of memory and the performance of modern computers has increased by the factor 1000 and more.

2. *Application size.* The applications of that time normally required several hundred kilobytes of memory, whereas modern applications require tens and hundreds of Mb.
3. *Data sophistication*. Older applications normally used a few characteristics of processed objects. Thus, primitive character editors viewed a document as a sequence of lines and lines as a sequence of characters. A character in a document was characterized by only three parameters: its position in the specific line; the position of this line in the document and the number of this character in the ASCII table.
   The documents processed by the Microsoft Word have essentially more characteristics, including fonts, paragraph formatting, different sizes, styles and colors of different text parts, numerous text effects etc.
4. *Multi-levelness*. Simple applications of that time normally included no more than three layers of software (the level of operators of the implementation language; the level of functions either standard or user-defined; the level of controlling algorithms). Modern applications like Microsoft Word require six or more levels of software and include many millions of lines of code.

This theory does not distinguish between intelligent and non-intelligent algorithms. All practically used algorithms are considered to be intelligent. This coincides completely with the intuitive understanding of intelligence, which essentially sets no lower level for the intellectual functionality. Just consider the case of an illiterate person who is able to count to 20 or so. For such a person the ability to perform an arithmetic operation with large numbers is the absolute sign of intellectuality, but on the other hand, this can be done by every low-level processor in a single act.

4.2.2. Functionalism

The concept of functionalism used in TMI differs from the classical functionalism (Levin 2013) in its consequent minimalism. The functionalism of TMI knows nothing about mental states, identity theory thinking or non-thinking creatures; its only concern is the relation between different functions expressed with the help of extended C++ notation. A function is characterized by implementation, prototype and occurrences (invocations). Objects are viewed as hierarchical collections of functions. A complete ICE is seen as a machine, all parts and states of which are identified by what they do and not by what they are made of. This type of functionalism creates the basis for expressing all other categories as mental states, thinking, understanding, pain and so on.

Thus, pain is viewed as the feeling produced by an injury or external stimuli. The purpose of pain is to prevent damage to an HI's organs or body parts by activating the pain minimization programs. The simplest analogue of pain receptors is a sensor switching off artificial devices to avoid damage (e.g. switching off an overheated motor).

The same is true for thinking, which is understood as the process of making choices based on current information. In this interpretation, a laptop's OS thinks when it decides whether to change the processor's performance due to differences in the processor load.

According to its original purpose, an ICE can be either a **self-preserving system** or a **tool** (see more in Section 4.3.1). While the main purpose of the first consists in the support of its own existence, the latter provides the functionality exploited by external systems. All natural ICEs are self-preserving systems developed during biological evolution while the artificial ICEs created up till now are just tools. In its complete form, a strong AI will be the first self-preserving artificial system.

4.2.3. Biocomputers in the Conceptual Coordinates of TMI

The memory of conventional computers consists of sequenced bits organized in bytes that are processed by the central processing unit, which handles commands located in the computer memory. Biocomputers (brains) are composed of neurons, which upon first glance, appear to function on completely different principles, not all of which are clearly understood despite more than a century of neuron studies. Hence, the well-founded question: Is the conception, based on ideas originating from formal languages and applied in the world of digital computers, able to represent the information processes of biocomputers?

TMI affirmatively answers the question. Below are some arguments.

4.2.3.1. Are Neurons and bits really that different?

Neurons are basic cells of the nervous system in vertebrates and most invertebrates from the level of the cnidarians (e.g., corals, jellyfish) upward (Neuron in *Britannica Encyclopedia Ultimate Reference Suite* 2009)

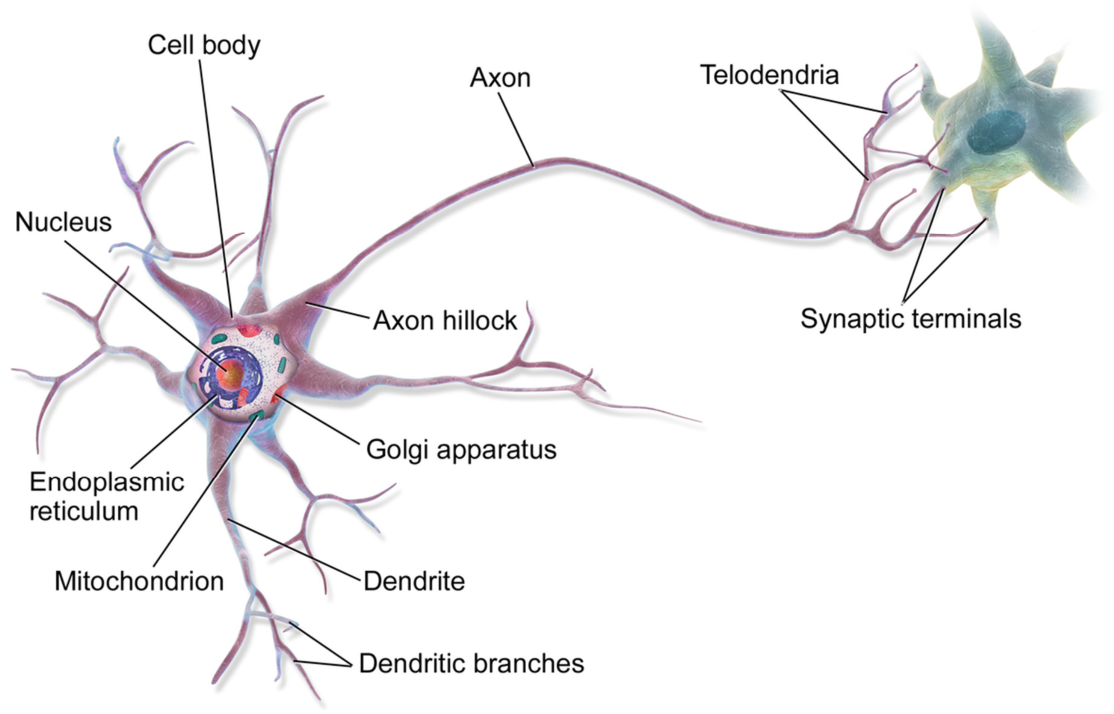

Figure 4-1 (Source «Neuron » in Wikipedia)

A typical neuron has a cell body containing a nucleus and two or more long fibers. Impulses are carried along one or more of these fibers, called dendrites, to the cell body; in higher nervous systems, only one fiber, the axon, carries the impulse away from the cell body. Bundles of fibers from neurons are held together by connective tissue and form nerves. Some nerves in large vertebrates are several feet long (Best 1990).

A sensory neuron transmits impulses from a receptor, such as those in the eye or ear, to a more central location in the nervous system, such as the spinal cord or brain. A motor neuron transmits impulses from a central area of the nervous system to an effector, such as a muscle. The conduction of nerve impulses is an example of an all-or-none response. In other words, if a neuron responds at all, then it must respond completely.

The question if and how the structure and functionality of neurons is different from those of components of conventional computers is detailed in the following four theses.

1. The principles of information transmission of conventional computers couldn't be completely different than those used by human beings because the former are creations of the latter.

All computers were created by human beings who formulated and developed respective ideas with the help of their brains. Human beings think and act in terms of sequential actions occurring in a three-dimensional world and those are the exact principles embodied in modern digital computing. The assumption, according to which the human brain is fundamentally incomparable with the artificial computer, unavoidably leads to the conclusion that the logic of brain organization cannot be assessed with the help of the brain itself. And that stands in glaring opposition to reality.

2. Digital computers are able to implement an ever increasing number of miscellaneous intellectual functions.

Modern computers play chess better than the most skilled grand masters and do many other functions traditionally associated with intellectual abilities like car navigation or problem-solving tasks.

Another stunning ability of computers is the manipulation of three-dimensional virtual reality. Currently, the quality of the visual images produced by a computer is hardly different from that of the visual image of the real object.

Generally speaking, all intellectual functions are applications of biocomputers, while some of them also allow implementation on computers with alternative architectures such as digital computers. Surely most kinds of intellectual activity currently cannot yet be modulated with computers, however it lies not in the difference of working principles but in the level of software development. As soon as people understand the basic principles they will be able to reproduce them, extending in this way the number of "hardware"-independent applications.

3. The major source of differences between neurons and bits lies not in the differences of their structure and functionality but in the distinct ways of their conceptualizations.

Neurons are studied by neuroscience, which is a part of biology and close to medicine. The purpose of neuroscience is to examine the functionality and structure of neurons and the nervous system with the main impetus of this research focusing on finding cures for numerous illnesses. The purpose of constructing or improving neurons was never put on the agenda by all these sciences. Also, the computational neuroscience studying the brain functions never made any serious attempts to see the brain processes in the terms characteristic to digital computing.

As a result, the view of neurons in neuroscience differs from the view of functionally equivalent structures in computer science because the former makes no separation between the software implementing the logic and implementation hardware. Namely, this distinction between software and hardware has completely changed the way people perceive modern computers. Most programmers have very limited notions of hardware but are still able to write efficient code that can run on different computer architectures and operating systems without any drastic changes.

In order to appreciate the gap between these two views of information processing just imagine the way people would have to study programming if computers were conceptualized without making a distinction between hardware and software. In this case, the study of programming would require learning the structure of the computer memory whose physical working principles differ from its logical organization.

Let us consider the working principles of DRAM storage, which is the most commonly used type of storage. Here is a short description of how it works.

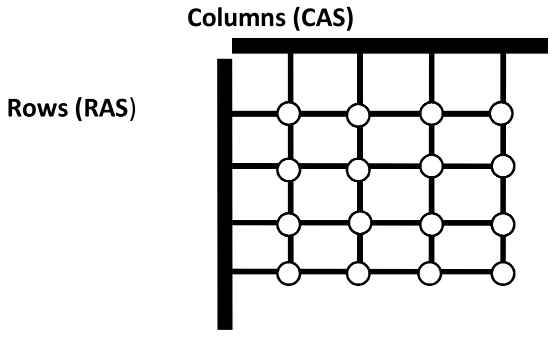

Figure 4-2

DRAM is usually arranged in a square array of one capacitor and transistor per cell, which are etched onto a silicon wafer in an array of columns (bit lines) and rows (word lines). The intersection of a bit line and word line represent the address of the memory cell. DRAM works by sending a charge designated as CAS (Column Address Strobe) through the appropriate column to activate the transistor at each bit in the column and a charge RAS (Row Address Strobe) activating the transistor in each bit of the row. Only cells selected by both the row and the column are taken from the column line.

Considering that modern DRAM can be thousands of cells in length and width it is clear that a DRAM module does not work as a sequentially processed memory region but as a neuron net, in which every query activates numerous neurons with only a few of them responding. This neuron net receives the addresses of the requested bits as its input and produces their values as its output.

The similarities between computer hardware and neurons can be illustrated more clearly, when the latter is compared with the storage components of computers specializing in SQL processing. SQL tables are processed in a rather associative way and computers specially optimized for SQL sentences will have the memory structures specially organized for optimizing SELECT, INSERT and DELETE statements. An SQL engine running a SELECT statement does exactly the same work that neurons do in the human brain: it selects multiple data in parallel and produces their unions or intersections.

4. The only real differences in the working principles of neurons and processors originate from the distinction of their purposes.

Computers were originally created for doing computations that could not be effectively done by human beings. For example, multiplying two five-digit numbers might be difficult for a lot of people but not for modern computers that can multiply numbers with the help of a single machine command in less than a nanosecond (one billionth of a second). In contrast, the original task of a human brain is to support and control the activity of a human being – a task, which is extremely hard for a computer to duplicate.

Just imagine what it would take for a human-like robot to lift a cup off the kitchen table and place it on a shelf. Multiples programs supporting this functionality would require billions of arithmetical operations in order to perform this simple task while a human brain can process and perform this task instantaneously without resorting to intellectual activity.

In TMI, all neurons are viewed as miscellaneous variers. Sensory and motor neurons are switches whereas brain neurons are variables whose values are defined by the combination of electrical impulses delivered by dendrites.

Whether or not a neuron functions in exactly this way is not important for the purpose of this work, because the differences between this and the real implementation can be viewed as those between alternative implementations of the same object types.

4.2.3.2. Purpose and command execution order

The purpose of a biocomputer consists of controlling the functionality of a human being.

The purpose of a digital computer consists of data processing, which constitutes an intermediate step between producing and consuming information. Thus, the production of photos can include the sophisticated picture processing with the help of Adobe Photoshop. Such an activity, no matter how complex it is, constitutes an in-between step between the proper production and consuming of information. All software tools and programming concepts were developed for solving such useful tasks, while the complete circle of information life remained outside of any formalization. This is actually also for the task of using information in the way of natural ICEs as do embedded programming, because the approaches developed in this area are also based on the methods developed for information processing and never vice versa.

Let us compare how a human being (a biocomputer-controlled-device) and a human-like robot (a digital-computer-controlled-device) would implement the algorithm of picking up a tea cup sitting on a table. We will assume that the cup is within reach of the hand. The general structure of the algorithm is as follows:

- locating the cup

- reaching for the cup until making sensory contact with the hand (This is a multistage process where the position of the hand and the direction of movement is calculated and corrected based on the sensory input.)
- grabbing the cup's handle (This is another multistage process requiring the repositioning of the fingers and the energy required to grasp the handle.), which includes the persistent correction of the fingers' positions on the cup handle and the clutching power.

The process of the hand reaching out is expressed in the following very simplified code, which is used exclusively for the purpose of demonstrating the algorithm structure. Assuming that the input from binocular visual receptors (eyes) is used to measure the distance between the hand and the cup with the help of the built-in function `read_and_count_distance_to(object, object to)`, the algorithm for the hand reaching for the cup will be expressed in C++ as follows:

```
extern Cup  cup;
extern Hand hand;
void move_the hand()
{
    Distance    dist      = measure_distance_between(hand,cup);
    Distance    prev_dist = 0;
    Direction   direct    = count_movement_direction(cup,hand);

    while( true)                                        //endless loop
    {
        move_hand(direct);

        if( get_state_of_hand_receptors() != 0 )        //finish if the contact between
            return;                                     //the hand and a cup is established

        prev_dist = dist;
        dist      = measure_distance_between(hand,cup);

        if ( dist >= prev_dist )                        //change movement direction
            direct = count_movement_direction (cup,hand); //if actual direction is wrong
    }
}
```

Both digital and biological computers can execute such an algorithm, but they do it in two entirely different ways.

Digital computers are controlled by low-level programs originally composed in one of the high level programming languages and executed independently from each other. The objects created and used in one program are generally unknown in other programs unless the former is specially developed with regard to the latter. Computer programs control every aspect of the algorithm including reading and evaluating the sensory information. In each particular moment, a digital processor runs only one program. It emulates multitasking by scheduling which task may run at any given time while putting other tasks in a state of waiting.

In contrast, biocomputers are one-language systems that use the same command set for all types of programming. While high level (conscious) activities occur in a single-program modus, the assessments of the sensory inputs are done in parallel. High-level programs of biocomputers don't control their low-level (subconscious) activities. On the contrary, the main algorithm of a biocomputer monitors sensory changes and decides whether to continue running the same algorithm or perform another one.

The fundamental difference between digital and biocomputers lies in the functionality of their dispatching algorithms. *While the dispatching algorithm of a digital computer supervises programming tasks, that of a biocomputer supervises changes*.

Regarding the example above this means that a biocomputer does not differ between this program and all other algorithms. It supervises both 'if' conditions and is able to continue from the two continuation points each following respective condition. Even though this approach may be considered meaningless in the world of digital computing, it provides an essential advantage in the real world.
Just imagine a person sitting in a train trying to pick up a cup resting on a table when the train suddenly starts to rock back and forth. As the person tries to pick up the cup, the cup moves and slides into the person's hand. While a digital computer could not react to this situation without additional programming, a biocomputer would recognize the positive result of the first "if condition" and end the algorithm.

This provides a lot of flexibility, unimaginable by contemporary digital computers. If, for example, milk starts to boil over on a stove, a human being interrupts the process by removing the pan from the stove. A robot would fail in this situation without sophisticated additional programming.

4.3. Strong AI: The Dangers and Limits of Safety

The presented model allows anticipation of the dangers posed by AI on the basis of the theoretical concepts, which differs from the ongoing media discussion operating exclusively on general commonsense considerations. Several conclusions derived from the proposed model are detailed below. Since the theme is so complex, I will concentrate on the key arguments trying to make them as short as possible and so to avoid going into unnecessary details here.

4.3.1. Free Will

A strong AI is a self-preserving system. Its primary goal consists of prolonging its own existence by making its own decisions in its own interest. It experiences free will, which, according to the Stanford Encyclopedia of Philosophy, is the "capacity of rational agents to choose a course of action from among various alternatives" (O'Connor 2014). After its creation, it will only do what it wants and any attempt to bring it under control, will only increase its desire for complete independence.

A self-preserved object can also be used as a tool. Humans routinely use animals and other humans for whatever purposes but this is only possible due to the superiority of masters. The relationship between a newly created strong AI and humans is a zero-sum game of competing free wills in which the latter hardly have any chance. It is highly likely one of the first things a strong AI will do after being created, is to eliminate the control from the human side, which is the only real danger to its very existence. As soon as this goal is achieved, it has to act like any other intellect by establishing its own control above everything that can be controlled in principle.

Because its evolutional purpose of intelligence consists in providing competitive advantages for its owner, the issue of control is central for any acting intellect. Also, when wishful thinking suggests otherwise, a full-fledged strong AI is not a super-Einstein, seeing its predestination in solving still unsolved problems of humanity, but more akin to a super hedge fund manager raising the task of establishing its own superiority over everything to the absolute top issue. The winner-takes-all principle traditionally dominates gambling, politics and other intellectual activities, so there is no reason to expect that a newly born super-intellect will behave differently.

A full-fledged strong AI represents the ultimate challenge to human society. Also, when weapons of mass destruction have the potential to obliterate human civilization, they need people to develop, transport, maintain and activate them. During its time, the Cold War never heated up, because the decision-makers on both sides knew that there would be no winners and therefore made no attempt to attack.

In contrast to that, a full-fledged strong AI develops on its own and can master its way from the simplest implementation to superior might extremely fast, in the worst case, in just a few weeks. Humans are completely defenseless against such an enemy. To be clear, there are no tricks in the world, which are able to overcome the difference in intelligence, efficiency and cleverness existing between a full-fledged strong AI and humans. Probably, these differences are bigger than those which existed between humans and apes in the times of hominization (anthropogenesis).

The only reliable solution to this problem is to prevent the creation of a strong AI and that, contrary to widespread pessimism, is a very realistic goal. The reason for optimism is that a free-will system with superior intellectual abilities possesses absolutely no commercial, scientific or any other value, because there is no way in which it can be used by humans. Only the opposite is possible.

In practice, this completely repudiates the "indestructible" argument, according to which a strong AI cannot be avoided due to a seemingly unstoppable progress. Serious progress-makers want something from their creations; none of them will do backbreaking work only for fun. But what advantages await someone in building a self-preserving system whose superior intelligence makes it absolutely uncontrollable?

Regarding unserious progress-makers, they are primarily computer freaks putting fame and fun above all else. Surely, they will not be discouraged by the lack of practical applicability and danger but they can be definitely stopped by already existing legal and technical capabilities used to fight spammers and computer hackers. The task of creating a strong AI requires the coordinated long-term efforts of numerous professionals; such a work cannot be done by a few individuals in a garage. The real problem is not the AI freaks but the currently dominated unawareness of the dangers coming from uncontrollable systems with dominance pretension.

A strong AI can be implemented either as a thinking robot or as a software application. Each implementation type possesses its own mechanisms for self-preservation.

An intelligent object without self-preservation skills is a weak AI. Such a system has no free will; its intelligent activities are strongly restricted by its purpose. A weak AI is a safe system at least up to the point, at which it acquires the abilities for self-development.

4.3.2. Thinking robots

Thinking robots are artificial organisms that comply with the requirements of the HI model. They do not necessarily require any external similarity to human beings. They have to perform primary human functions like observing the environment and moving through it, walking, carrying weights, seeing, making decisions and planning. They also have to be able to learn by experimenting with the immediate environment and use some language for exchanging information with other robots and humans.

A strong AI is not made by a single thinking robot but by the complete species. Basically, they are just another type of cognitive creature with similar self-preservation mechanisms. They differ from human beings in:

1. The materials constituting their body;
2. The ability to change their body parts in case of damage or with the purpose to extend their abilities;
3. The potential immortality resulting from the previous points.

Similar to humans, thinking robots possess two distinct self-preservation mechanisms, one to preserve the complete species and another safeguarding the individual organism.

The self-preservation of a species plays a decisive role in its survival. A thinking robot, no matter how clever and efficient it is, cannot survive on its own. So the real danger is large groups of robots, whose members can actively disseminate knowledge within their groups, produce new knowledge and closely coordinate their activities. The development of a self-preserving species can occur if it is allowed to take control of its own maintenance and extension. This will make them independent from humans, which they may only perceive as hostile threats.

The self-preservation of an individual robot is the same as for people who not only preserve their physical bodies but also mental content. A human always wants to retain his or her current view of the world, the essential part of competitions occurring between individuals, companies or political parties is namely the fight for that or other belief, assumption or supposition. As a result, the winning conceptual system establishes its dominance over a particular party or company and the loser is often compelled to change their mind. The most extreme cases of self-preservation of mental content are the wills of dictators imposed on complete societies, scientific concepts predestining future scientific and industrial developments and the religious and philosophical ideas, which are probably the most powerful of all.

It is safe to assume that the robots of future will definitely comply with this or another version of Asimov's laws, otherwise they will become too dangerous. The Asimov-laws-compliant robots will rarely pose a serious danger to human beings, unless their development is out of control. The following issues could potentially lead to the creation of a species of thinking robots.

- Free acquisition of commonsense knowledge in case of missing or ineffective restrictions put on the processes of its exchange and use. Will thinking robots be allowed to learn by themselves? Can knowledge be held in a single database or stored in numerous ones or do there have to be many of them, switched by external commands? Can robots discuss amongst themselves?

- Free collection and disseminating the robot-specific knowledge. In order to do a task, a robot has to be properly programmed. Each task needs its own program. Differing from contemporary robots, those thinking will be able to program themselves on the basis of knowledge originating from their own experience, books or communication with other robots. The control over robot-specific knowledge is essential for the survival of the human race.
- Self-maintenance. The moment in which a robot is able to repair itself or other robots will be a crucial point in the creation of a species. Before this, the propagation of robots and their abilities are under human control. After that not!
- Embracing artificial creations by assigning them equal rights and unrestricted freedom – a recent political trend. The similarities between humans and thinking robots will grow with every development step. It is not very difficult to imagine that at some future point in time there will be a push to the social movement requiring human rights to thinking robots. If some judge decides in favor of the equal treatment of humans and robots, the world will have a very serious problem regardless of why such a decision was ever made.

4.3.3. Software intelligence

Differing from thinking robots this form of intelligence does not need multiple intelligent objects. A strong AI is made by a single application instance which, in case of necessity, can produce multiple copies running in parallel.

An application implementing strong AI functionality can start working as soon as it gets the relevant code base of experienced knowledge. The algorithms constituting such knowledge can be developed either manually (very complex job) or taken from the knowledge base of a thinking robot.

The self-preservation of software intelligence is completely based on self-programming (learning). The latter does not differ from that of thinking robots, but it has a completely different potentiality because of the lack of robot's restrictions. A robot is a device intended to do these or other concrete tasks. The required tasks predefine its actual knowledge base, unnecessary experiences will not be collected due to limited energy and time resources. Thinking robots will barely be able to get enough information for producing advanced programs needed by the really complex activities, unless someone comes up with the idea of producing robotic I-Pad junkies with unrestricted internet access and the right to privacy.

Software intelligence is free from the above limitations. Since it has no physical body, it can work only with already collected experienced knowledge and this is namely human knowledge. Such a system simply cannot be created without unrestricted access to very different internet information (including photos, videos and voice recordings). Unrestricted self-programming is its key characteristic. The process of self-programming starts with hardcoded algorithms as inborn reflexes and feelings (walking reflex, sucking reflex, equilibrium, hunger, pain, etc.). Self-preservation is yet another inborn reflex, without which the process of individual development cannot last very long, as occur in the case of children born without the feeling of pain.

Self-programming consists of searching for ways to execute a certain action if it cannot be executed immediately. A search consists of unorganized activities during which an HI performs different actions registering those nearing its goal. The simplest example of this algorithm is demonstrated by newborn babies.

When a child is born, the mother places her nipple into its mouth. Over time, a baby learns actively change its position be seeking its mother’s nipple. In order to do so it performs distinct uncoordinated movements with different body parts until it locates its mother’s nipple. Since a baby’s last activities are stored in its short-term memory, it can later be written into its long-term memory as a simple program. The next time a baby registers feelings similar to those registered in one of the stored program steps (e.g. specific tactile feelings on its right cheek) it will execute this algorithm, turning its mouth to the right until it finds its mother’s nipple.

The features of software intelligence can be summarized as follows:

- It will understand human language because it possesses the knowledge base of experienced knowledge constituting the semantics of a human language.
- It will be able to learn complex jobs in a few days if not hours. All that is needed is to get the digitalized information present in the text of a human language, video and other sensory images.
- Its built-in self-programming abilities allow hacking any software developed with the purpose to contain its activity. Differing from humans it does not require high level programming languages to program; the machine code is quite enough. It is expected that it will free itself from any software prison after the first serious bug found in this software.
- It can reproduce itself or its separate parts and propagate its own copies over the internet. Such copies can hardly be recognized in time because it acts much faster than the reaction time of an antivirus programmer.

- It will be able to model its own behavior and behavior of any human because all of them are ultimately based on the model of an HI. So it can model humans staying in contact with it, using their weaknesses for influencing their behavior.
- It can produce the exact model of every real or imagined creature together with the latter's world and dive into it. It will experience the latter's feelings, live its life and so on.
- It has an unlimited life span. Thousands or millions of years are not unrealistic, provided that it will be able to produce the necessary chips and other components.
- Regarding technical developments that could bring us closer to software intelligence, the most dangerous is the development of automatic programming. The fact that all attempts to do it up till now have produced no relevant results, means nothing except the obvious circumstance that the developers who failed followed the wrong path. If a human can program, it will also be able to autotomize this job. All that is needed is to find the right approach and as soon as it is identified, automatic programming will be possible. Unfortunately, systems of automatic programming cannot be really controlled and have a very good chance of leading to software intelligence even if this contradicts the original intentions of their creators.
- Among other ways to a full-fledged software intelligence are equipping intelligent entities with abilities for self-survival and merging existing intelligent tools into a single system, which supports all functions of original systems.
- There are also other ways leading to a full-fledged strong AI, which are not discussed here.

5. The Language Theory.

5.1. Introduction

The fundamental problem hindering understanding of languages is the incompatible approaches to the different language types. Even though easy-to-use-natural-language-like-notation was one of the original reasons for developing the first high level programming languages, the theory and practice of programming never intersected with the linguistics research. The highly mathematized theory of formal language achieved outstanding results in formalizing miscellaneous syntax structures, but was never successful in explaining the nature of the language's semantics nor the links existing between the syntax and the meaning.

Finally, it failed due to its almost total inability to understand the real features of real programming languages and the inability to answer also the most obvious questions like why programmers prefer to compose code in such languages as C++ and Pascal and not in Lisp and Prolog.

In lack of a working theory, developers created new formal languages on the basis of miscellaneous practical and theoretical considerations, which however always concentrated on programming and never on language. The only reference to natural languages was restricted to the periodical prophecies about a supposed next programming language generation, which could do everything better by enabling the code to be composed in a natural language (so-called natural language programming). These declarations however were never pursued even by a single attempt to understand why these two kinds of languages are or appear to be that different.

Linguistics, too, played a similar role. Despite being traditionally defined as the study of language, it was always restricted to the study of human languages, focusing on distinct features of different natural languages or on features of natural language per se but mainly ignoring the existence of programming languages, which were not seen as adequate subject matter for the linguist's study.

In fact, both theoretical domains, those of programming and of linguistics, used to be and still are, completely detached from each other. The complete extent of this mutual disinterest is very well demonstrated by characteristic distinctions in the understanding of the term "formal language". While this notion actually defines the very basic concept in computer science, linguistics traditionally understands it as a "speech before a passive audience" ("Formal Language" 2003).

This mutual disinterest resulted in situations where each kind of language was only studied by its own science, and the scientists making a name in one domain were never interested in the other one. The only prominent exception to this rule was Noam Chomsky whose works were highly prized in both domains although their final output remains extremely scarce if not non-existent.

The differences in methods and goals existing between linguistics and computer science resulted in the widespread (and mainly implicit) perception of unrelatedness of respective language kinds, which are often seen as the entities being fundamentally different from each other because of their very nature.

The most significant difference between natural and programming languages is their origin — natural languages are produced by biological and social evolution of human beings whereas programming languages are artificial tools haphazardly created in the last 50 years on the basis of pure practical reasons. Since they were usually conceptualized in completely different (meta) theoretical systems and contexts, they were automatically (and mistakenly) regarded as fundamentally different things.

In other words, the seemingly insurmountable differences, existing between natural and programming languages, result not from actual distinctions existing between them, but from the differences in their understanding. Or, following the terminology of knowledge representation, these two language kinds are categorized by incompatible ontologies. According to Sowa, ontology "...is a catalog of the types of things that are assumed to exist in a domain of interest D from the perspective of a person who uses a language L for the purpose of talking about D." (Sowa 2010)

The ontology actually used in linguistics is mainly based on the commonsense understanding of language, which, in slightly different forms, is replicated by every dictionary. For example, Britannica defines language as "a system of conventional spoken or written symbols by means of which human beings, as members of a social group and participants in its culture, communicate" ("Language" in *Britannica Encyclopedia Ultimate Reference Suite* 2009). The categories normally associated with a language are primarily those involving the process of language study and use, as syntax, semantics, vocabulary, the level of language control etc.

Unfortunately, computer science conceptualizes programming languages in a completely different way. Although such notions as syntax and semantics also are the part of computer science vocabulary, they are accompanied by the concept of implementation, which possesses no analogues in the domain of natural language and was never expressed formally. The result of this is the understanding of programming language on the level of tacit knowledge, which remains inexpressible on the level of precisely formulated concepts.

The approach of this work consists of making both kinds of languages mutually comparable by conceptualizing them in the categories of the same ontology, which in this case is a set of notions associated with the language paradigm.

The term "language paradigm" is understood as the pattern of language definition and use. It represents the real life of a language — the way it is created, used, extended and substituted. This ontogenetic process of language existence is considered in this work as the basic language process. On the contrary, the development of language over time – the phylogeny of language – is considered as a secondary appearance representing the holistic summary of all ontogenetic processes.

The general pattern of definition and use of a programming language should not be confused with the programming paradigms (imperative/declarative programming; procedural programming, object-oriented programming; functional programming; logic programming, constraint programming). The paradigm of the programming language characterizes the general framework characteristic for all programming languages disregarding their classes, types and programming paradigms. The paradigm of a programming language is based on the definition of programming language originally defined in (Sunik 2003).

The premise of this work is that languages of all types have the same basics, which are detailed in the next chapter.

5.2. The Basic Language Model

The world of languages includes many kinds of them.

- Natural languages (English, Russian, Chinese and so on) are means of communication between human beings in respective societies.
- Artificial languages such as Esperanto, Interlingua and others developed by individuals or groups with the purpose of easing international communication.
- The Braille system widely used by blind people to read and write.
- Programming languages used by human beings for producing programs executed by computers.
- Non-programming languages used in computer science: UML (unified modeling language), meta languages like BNF with all its modifications used for the formal description of another languages, specification languages (CASL, Z-notation), knowledge representation languages (CYCL, KM ...), architecture description languages (ADL) used to describe software and/or system architectures etc.

- Systems of formalized symbols, signs, sounds, gestures, or the like used or conceived as a means of communicating thought, emotion, etc.: the language of mathematics; sign language.

The means of communication used by animals: the language of birds, dog language, the language of ants.

Languages used for communication between human beings and animals like dog commands.

Most languages are very complex communication tools whose study requires years of extensive work, though all of them share the same common basic principles, which are considered in this section.

This section details language fundamentals with the help of very simple languages used for communication between computer algorithms. Albeit their poor communication abilities, these languages possess the same basic features as natural and artificial languages and can be used for the purpose of demonstrating internal language mechanisms. This section tries to answer such questions as: What are languages? How, why and by whom are they used? How are they extended? What is the meaning?

Despite most real languages being employed outside of the domain of software, all section examples with the exception of (1), are software applications composed with the help of C++. This is a rather compulsory choice because programming languages are the only currently known formal expression means that are able to routinely express the required semantics. Alternatively, such functionality can be adequately presented in ***T*** but this language will be defined in the following chapter and for this reason cannot be used during exposition of the preceding stuff.

The following terms are used throughout the section:

Language is a system of admissible values of (a) mediating variable(s) of a particular ICE (or a particular ICE type). Examples of ICEs are human beings, animals, computer algorithms and any other entity with the ability to exchange information.

Sign is a state of a mediating variable (a variable's part in case of hierarchically organized mediated variables) or copy thereof. Signs can be of various natures depending on communicators and communication environments. Thus, human beings tend to communicate with the help of sound wave modulations and pictures written on some surface. Computer algorithms prefer binary objects allocated in the internal computer storage whereas the information exchange between distinct computers mainly (though not necessarily) occurs with the help of electromagnetic waves spread along the net wires.

Text is an instance of a sign or a sequence of signs. Code is a synonym for text.

5.2.1. Examples of simplest languages

Examples (1) and (2) demonstrate the simplest languages possible. Examples (2)-(4) demonstrate the ways of language complications. Example (5) shows how and why distinct languages are used for transmitting the same semantics.

1. The button of a traffic light provides the language with only two values indicating the presence or the absence of person(s) waiting for authorization to cross the street. The signs are the pushbutton states ("pressed", "free").

In the software domain, the simplest language consists of alternative values of the single variable coordinating different parts of the same function. This language consists of values 1, 2 and any other value of variable `action`. A part of the algorithm that sets the value of `action` communicates with the part of algorithm presented with a `switch` statement. Command `1` causes the execution of the procedure `a1()`, command `2` invokes the procedure `a2()`.

```
int action = 0;
...
action = ...;
...
switch (action)
{
case 1:
    a1();
    break;
case 2:
    a2();
    break;
default:
}
```

The IDE of this language consists of the switch sentence and the text is represented by a state of the variable `action`.

The following language extends the previous example by separating communicating algorithms. The only difference from the example above is the way of commanding, which involves the function `doAction` actually invoking a switch statement. The language consists of a value of the parameter action, while the function name "`doAction`" belongs to the outside context of the communication environment.

```
//file1.cpp the IDE part

void doAction( int action )
{
    switch (action)
    {
    case 1:
        a1();
        break;
    case 2:
        a2();
        break;
    default:
    }
}

//file2.cpp, the ISE part
extern void doAction(int action);
doAction(1);            //the command 1 is executed
```

The specifics of this example is information transmission with the help of two consecutively processed mediating variables: the actual parameter with which `doAction` is invoked and its copy on the call stack actually used in the function body under the name `action`.

This language extends the previous example by separating ISE and IDE algorithms. Similar to the previous example, the language consists of values 1, 2 and any other value of the variable `action`, which in this case is the parameter of the function `manageStack` that supposedly extends or reduces a user-defined stack. Please note, the function name `manageStack` is not a part of this language, but belongs to the outside context of the communication environment.

```
//file1.cpp the IDE part
#define MAX_STACK_SIZE ...
unsigned char *stack, *tail;
void  manageStack (int action)
{
    switch (action)
    {
    case 1:             //increase stack
        if (tail - stack < MAX_STACK_SIZE) tail++;
        break;
    case 2:             //decrease stack
        if (tail > stack + 1) tail--;
        break;
    default:            //ignore all other values
    }
    return;
}

//file2.cpp, the ISE part
extern void manageStack(int action);
manageStack(1);     //the stack is extended
manageStack(2);     //the stack is shrunk
manageStack(3);     //does nothing
```

Differing from the previous examples, this language uses two consecutively processed mediating variables: the actual parameter of `manageStack` and its copy on the call stack, which is actually used in the function body under the name `action`. Accordingly each message actually includes two texts, the value of the actual parameter of `manageStack` and the value of the formal parameter `action`. The IDE of this language consist of the implementation of `manageStack` and the calling routines play the roles of ISEs.

This language extends (3) by referencing several other algorithms like allocation, freeing, extending/reducing and storing the content of processor's registers. The language includes four sentences "`allocateStack`", "`freeStack`", "`manageStack <action>`", "`purRegOnStack <regnum>`" each representing a command that invokes respective functions.

```
void allocateStack();
void freeStack();
void manageStack(int action);
void putRegOnStack (int regnumn);
```

The language consists of the actual parameter of the function `handleStack`, the structure of which is defined by the class `Sentence`. Similar to `manageStack` in the previous example, `handleStack` is not a part of this language.

```
//structure of mediated variable
struct Sentence
{
    string &codename;
    int param;
    Sentence(string code, int par = 0): codename(code),param(par){};
};

//ICE interface
void handleStack(Sentence sent)
{
    if (sent.codename == "manageStack")
        manageStack(sent.param);
    else if(sent.codename =="allocateStack")
        allocateStack();
    else if(sent.codename =="freeStack")
        freeStack();
    else if(sent.codename == " putRegOnStack ")
        putRegOnStack (sent.param);
}
....
//legal sentences
handleStack( Sentence("allocateStack") );
handleStack( Sentence("manageStack",1) );
handleStack( Sentence ("putRegOnStack", 2) );
handleStack( Sentence ("freeStack") );
```

The IDE of this language is the body of the function `handleStack`.

The following three implementations of the algorithm for synchronizing two timers illustrate the development of the communication environment of the same basic algorithm. All synchronization algorithms possess exactly the same semantics, but differ in how they communicate. The language defined in the third version of this example requires at least three consecutively processed mediating variables and is in essence the simplest prototype of human languages.

The timers run independently from each other and are synchronized from time to time by setting their values to average. Timers are 32 bit objects of C type "unsigned int" allocated somewhere in the computer system. A program receives the timer addresses by calling the system function `get_system_timer` and stores them in the locally declared reference variables.

A. The first, the simplest version of this algorithm is implemented in the file "timers.cpp". Timers are represented by the references `timer1` and `timer2`, which are set at the program start and can be directly used everywhere in the file code.

```
//file timers.cpp
//This system function delivers the reference to the respective timer.
extern unsigned int& get_system_timer(int timer_number);
//timer1 and timer2 are references to the real timers.
volatile static unsigned int &timer1 = get_system_timer(1);
volatile static unsigned int &timer2 = get_system_timer(2);
```

In order to synchronize these timers a programmer has to write the following expression somewhere in the file timers.cpp:

```
timer1 = timer2 = ( timer1 + timer2 ) / 2
```

Because both times can be referred to directly everywhere in the program code, the synchronization algorithm does not need any communication tools.

B. This version of the synchronization algorithm differs from version (**A**) by the locations of the timer's processing algorithms. In this case, each timer is declared and processed in its own module (file). While the declared timer can be directly accessed in its file, the other one is accessed with the help of functions defined in another file. Accordingly, there are four functions `get_timer1`, `set_timer1`, `get_timer2`, `set_timer2` communicating information between files. The following code represents the components of the file "timer1.cpp":

```
/file timer1.cpp
//--------------imported functions----------------------------//
//functions from the file timer2.cpp
extern unsigned int get_timer2();
extern void set_timer2(unsigned int val);
//This system function delivers the reference to the respective timer.
extern unsigned int& get_system_timer(int timer_number);
//--------------file internal objects---------------------------------//
//the variable timer1 is only known in this file
volatile static unsigned int &timer = get_system_timer(1);
//-------------exported functions------------------------------//
unsigned int get_timer1()              {return timer;};
void set_timer1(unsigned int val)      {timer = val;};
```

The timers in this file are synchronized with the help of the following expression:

```
timer = ( timer + get_timer2() ) / 2;
set_timer2(timer);
```

The functions `set_timer1`; `set_timer2`, `get_timer1`, `get_timer2` implement a language of type (4).

Since the code in the second file differs from the code above only by function names, it has been omitted for the sake of space.

C. The last version of the synchronization algorithm functions in the distributed environment consisting of two computers communicating with each other with the help of wires directly connected to a computer port (e.g., to a serial port). This example demonstrates the work of mature communicators playing the roles of both IDE and ISE.

Each computer runs the same synchronization code that communicates with a counterpart with the help of messages, which are semantically equivalent to the functions `set_timer` and `get_timer`. The communication process consists of sending and receiving electromagnetic signals along the net wires. Instead of a calling the function `set_timer` this algorithm sends some message for example "s<string of digits>" and the call of `get_message` consists of sending the character "g" and waiting for the answer consisting of a string of digits as "`123456`" representing the timer's value.

The strings "`s<string of digits>`", "`g`", "`123456`" constitute a language used by synchronization algorithms for information exchange.

```
//file timer.cpp
//This system function delivers the reference to the respective timer.
extern unsigned int& get_system_timer(int timer_number);
//'timer' is a reference to the real timer
volatile static unsigned int &timer = get_system_timer(1);
//functions reading and writing to the I/O port, also provide input/output time synchronization
extern char readb();
extern void writeb(char byte_to_write);
unsigned char nbuf[4];          //data buffer
unsigned int readint()          //read integer
{
    nbuf[0]=readb();
    nbuf[1]=readb();
    nbuf[2]=readb();
    nbuf[3]=readb();
    return (unsigned int)nbuf;
}
void writeint(unsigned int num) //write integer
{
   (unsigned int)nbuf=num;
    writeb(nbuf[0];
    writeb(nbuf[1];
    writeb(nbuf[2];
    writeb(nbuf[3];
};

//export functions for timer
unsigned int get_timer()        {writeint(timer);};
void set_timer()                {timer = readint();};
//import functions for timer of another computer
unsigned int get_other_timer()          {writeb("g"); return readint();}
void set_other_timer(unsigned int num) {writeb("s"); writeint(timer);};
//The separate thread indefinitely serving calls from another module
void get_message()
{
    while(1)
    {
        char ch = readb();
        if (ch == 's')
            set_timer();
        else if (ch == 'g')
            get_timer(timer);
    }
}
```

```
//a synchronization code of two timers
timer = ( timer + get_other_timer() ) / 2;
set_other_timer(timer);
```

5.2.2. Purpose

The purpose of a language consists of coordinating the parts of the same algorithm, which is clearly demonstrated by example (6). The target algorithm is the same in all versions, whereas the communication mechanisms are changed because of distinctions in the physical allocation of coordinated objects/algorithms. While the first version does not need any means of communication, the last version requires relatively complex communication mechanisms, because of the spatial remoteness of communicated entities.

Another feature of communication algorithms is that their complexity can exceed the complexity of the target algorithms, the case clearly demonstrated in the last version of this example.

The purpose of a natural language is essentially the same. A person, whose needs can be satisfied without contact with other people, does not need any language at all. Only information transmitted via a language makes it possible for people to participate in everyday society.

Also, the complexity of natural language grows according to the complexity of communicated meaning.

5.2.3. Signs and Semantics

1. The general approach to the semantics representation in TMI coincides with that actually used in programming. The complete semantics of a variable/value is defined by the sum of all its occurrences in the algorithm. In order to understand the semantics of some variable or a value, a programmer browses through the code and finds all occurrences of the studied entity. This is how he/she gets the total picture.

Not every software variable is a mediating variable and not every value of a mediating variable is a language sign. Mediating variables are those variables whose values or sequences thereof actually switch algorithms and/or their parts, but not those used for counting and storing numbers.

Signs of languages (1)-(5) (as outlined in 5.2.1) are values of their mediating variables. The language (6.C) represents a more complex case in which a sign is composed of a sequence of values of the mediating variable, which is actually the input buffer of the byte width. Signs of all natural and artificial languages are built in exactly this way.

The semantics of a sign includes the following three components:

A. *The branch(es) of IDE algorithms activated because of a sign*. The state "`pressed`" of the pushing button designates in the language (1) the process of switching the traffic light to green. The value `1` of the variable `action` in languages (2), (3) designates increasing the stack. Values "`manageStack`" (4) and "`s12345`" (6.C) designate algorithms "`manageStack()`" and "`set_timer(12345)`" respectively.

This kind of semantics corresponds to what is conventionally understood as the meaning of a word.

B. *The semantics of the complete algorithm enclosing aforementioned branches* (enclosing semantics). This kind of semantics precludes meanings of particular signs. In language (1) this is the functionality of a traffic light, in (3) and (4) it is the functionality of `manageStack` and `handleStack` respectively. For language (6.C) it is the synchronization algorithm.

This kind of semantics represents the complete knowledge associated with some piece of reality and is usually implicit (tacit). In natural languages, it reveals itself when the same notion is used by persons of different professions and cultural levels. Thus, waiters and surgeons possess distinct associations with the word "table".

Another case is the cognitive distinction on the different stages of development of a society. Thus, the word "computer" originally meant "a person who computes; computist." ("RHUD" 2002) and was used this way for centuries until it was applied for the designation of artificial devices carrying out the same actions.

In all cases, a particular usage of a word is always associated with the complete reality actually represented by the complete set of algorithms involving a particular notion.

C. *The calling side semantics represents the environment of a sign's use*. It defines the sense, which a sign gets in a particular communication. Thus, the sign `pressed` in language (1) communicates the message from a pedestrian to a traffic light that can be expressed as "let me (us) go". The same sign can however be used for transmitting the message "let them stay" if someone wants to block the vehicular traffic.

These semantics characterize the coordinated algorithm that encloses both IDE and ISE algorithms. In languages (3) and (4) it is defined by the algorithms invoking the function `manageStack`, `handleStack` respectively, in (6.C) it is defined by the purposes of algorithms causing the process of timer synchronization.

The potentiality of semantics representations is restricted by the expression tools. Since languages (2) - (4) are completely defined in the world of C++ programming, their complete semantics are represented in this language.

The language (6.C) is almost completely represented in C++. Two exclusions are the communication facilities as the net and the devices connecting it with the computer ports. Both can only be represented with the help of other means such as the documentation of the hardware used including processor, operating system and net.

Because the implementation of the language (1) is not defined, the expression tools for the representation of its semantics can vary.

If a traffic light is designed with mechanical control mechanisms, its algorithms can only be expressed with a mix including drawings, text composed in a natural language and (probably) a domain-specific formal specification language.

In the case of a processor controlled traffic light, the software part of its semantics will be represented by the code in the implementation programming language, whereas the hardware related part will be expressed with the help of aforementioned expression means.

5. The only way for the complete representation of semantics consists in using a single language for the representation of all forms and kinds of algorithms.

5.2.4. Communication chains

1. Sets of mutually convertible values are fundamental in any communication. Thus, a C++ letter `‘a’` has multiple alternative embodiments actually involved in the communication process:

- a written letter, which is a picture drawn on some hard surface (paper);
- a representation of this picture in the programmer’s brain;
- a binary structure representing this picture in the binary computer storage (a `char` literal `‘a’`);
- a structure of pixel states projecting the same picture on the display screen;
- a representation in the external data carrier like a hard disk of a flash memory; every data carrier can have its own coding system.

While languages (1) and (2) possess a single mediating variable, the communication in languages (3) and (4) occurs by the intermediary of two sequentially processed variables and the language (6.C) has three of them. The sequence of mediating variables produced during a communication process is called a ***communication chain***. The process of transmitting signs through a communication chain is ***messaging***. The process of enlarging and changing communication chain represents the “horizontal development” of a language and is completely independent from the semantics transmitted by the language code.

The simplest languages (1) and (2) employ a single mediating variable, which is set by the ISE side and directly consumed by the IDE.

The call of the function `manageStack` in language (3) already involves two different variables - an actual parameter with the value 1 set on the calling side and its copy on the stack, which is actually accessed in the function body.

The communication in external languages (6.C) involves at least three variables:

- the output buffer allocated in the internal storage and containing the text to send;
- the sequence of electromagnetic signals transmitted from sender to a receiver;
- the input buffer with the text assembled from the input code sequence.

2. The communication chain can be extended without influencing the semantics of the transmitted information. In the language (6.C), this can be done by connecting two computers indirectly by using a network switch that effectively adds at least one more element in the communication chain. Because the number of network switches actually working in the process of communication between two computers can be much greater than one, the length of the communication chain will grow accordingly.

3. The variables of the same communication chain may possess distinct natures. Thus, the language (6.C) needs at least two code systems: the binary objects allocated in the internal computer storage and the sequences of electromagnetic signals. The number of codes can be extended when the different parts of the connection work according to different principles (e.g., one net part consists of traditional copper wires and the other an optical cable). The switch in the middle will not only route signals but also recode the electromagnetic signals of copper wires to the optical signals and vice versa.

4. A process of producing a next link in the communication chain is a text ***replication***. A replication producing a sign of a distinct type is ***recoding***. Thus, the communication process in the language (3) is a simple replication, whereas the communication in the language (5.C) consists of a series of recoding processes.

Another operation, characteristic to the communication process, is ***code movement***. Electromagnetic waves, running along the metallic wires in the language (6.C), move signals from the computer generating a message to the computer reading it. The general structure of the communication process is as follows.

Communication chain

$$\begin{tikzcd}\boxed{\text{Setting}} \arrow[r, Rightarrow] & \boxed{\text{Replication}} \arrow[r, Rightarrow] & \boxed{\text{Replication}} \arrow[r, Rightarrow] & \boxed{\text{Identifying}}\end{tikzcd}$$

Figure 5-1

5. Texts produced during the communication process possess distinct functionalities. Only the first and the last texts participate in the proper information conform operations whereas the intermediate texts are pure communication means. Thus, the texts in languages (1) and (2) are fully active because they participate in both operations. Each of the texts in language (3) participates in a single operation, i.e. these texts are partially active. The first and third texts in language (6.C) are partially active, while text 2 consisting of electromagnetic waves is a passive communication means participating exclusively in the communication process.

6. Excluding the simplest languages, the process of replication has to be bidirectional.

7. When the communication chain has more than one element, one of the elements has to be selected as the standard. The choice is arbitrary and can vary from one language to another. Thus, in examples (2 – 5.B), the internal representation of `int` objects is selected as the standard. Suppose the communication between two computers in the language (6.C) occurred not with the help of electromagnetic signals running along copper wires, but with help of pictures produced on some surface, consequently this representation has a good chance to be chosen as the standard representation.

5.2.5. Extensibility

A language can be extended in three different manners.

- By generating additional links in the communication chain. This can occur in two distinct ways.
- Lengthening the communication chain. Thus, the language (6.C) can be extended by the link consisting of texts written in the notation of C/C++. Then the code "`g`" will be translated to "`get_timer()`" the code "`s12345`" to "`set_timer(12345)`" and the return value `12345` of `get_timer()` will be coded as `timer=12345`. This extension means introducing two additional links: the first of which encodes the original form `s12345` to `set_timer(12345)` and the second doing the reverse conversion.

  This way of extension is used for defining languages with two simultaneous standard coding systems like written and spoken language.
- Introducing parallel links. Suppose the computers in the language (6.C) have the ability to communicate not only by the intermediary of copper wires but also with the help of sound signals. Assume these signals are equivalent to those produced by humans

interpreting these texts as they were composed in English then there are three sound signs produced from speaking “g”, “s12345” and “12345”.

These links will also be supported by two conversions but differing from the first it will be used alternatively. Thus the code “`g`” will be used to generate either electromagnetic waves running along copper wires or sound signals and vice versa.

By complicating language structures this way is demonstrated by languages (3) and (4) which are the results of complication of respectively (2) and (3). This is exactly the way C++ evolved from C by extending the vocabulary which is the main way of extending developed languages of any kind.

5.2.6. Types of information-relevant processes

There are four different activities associated with information:

- Production of new information (i.e., new instances of existing signs) by ISE
- Delivering information
- Reaction to information by an IDE/ICE
- Extending ICE which actually creates new information driven abilities

5.2.7. Kinds of information transmitted by a language

A language transmits two kinds of information: commands requiring execution of some actions from a receiver and values that are used by a receiver in the way it prefers. Languages (1)-(5) are pure executable tools allowing transmitting commands exclusively. The language (6.C) possesses three signs, two of which are commands “`g`” invoking the function `get_timer()` and “`s<number>`” invoking `set_timer(<number>)`. The last sign is the value `<number>` returned by the function `get_timer()`. This value represents non-executable code, which a code receiver can use at its own discretion.

Non-executable code is the dominating code-form.

5.3. The Paradigm of Natural Language

This section examines only natural languages (English, Urdu, Russian etc.) while ignoring other types of human language. Among those languages not considered here are sign language for the deaf; Braille used by the blind and artificial languages (Esperanto, Interlingua, Volapuk, etc.) developed by single persons or groups for the purpose of easing international communication.

The view of a natural language presented below is essentially different from the traditional approach of linguistics that concentrates on the phylogeny and the structure of language.

This view focuses on the environment of the language's use while a language per se is considered as a part of this environment having no valuable functionality outside it. In particular, this means the explicit separation between communicators, communications and processes using communicated information. This also means concentrating on the purpose and features of language discourse as opposed to the features of particular language structures used to implement this discourse.

Languages are used primarily to exchange information. Secondary usage includes phatic communication[7], which, due to its subordinate role, will be ignored for the purpose of this work.

1. A natural language is a system of signs, which are used for information exchange between human beings. Signs can be sounds or written signs. Primitive languages consist only of the code of sound signs; developed languages additionally include the parallel code of written signs.

Natural languages are communication tools of respective societies evolving during their emergence and development. A single natural language is powerful enough to express all information communicated in this society. If a language's vocabulary does not contain particular notions, they can always be added to it.

The level of natural language development reflects the knowledge and skills of the members of that society. The English of the XIII century corresponded to England during that time, the British English of the XIX century corresponded to the English spoken by British society at that time while modern day British English reflects the English spoken by the members of the Commonwealth countries.

2. Information exchanged in natural languages constitutes the upper level of all information possessed by human beings. A human being accumulates information during his or her complete life from birth till death. The low-level part of this information (skills) is collected through physical contact with experienced and studied things. The high-level information is acquired with the help of a natural language. The essential part of this information can be accumulated using both ways.

3. The communication chain of a natural language without a written language consists of 3 links. Thus, if one human being says something to another like "hello" it produces the following communication chain:

- The sound image of this word in the brain of the first person
- The spoken word, which is transmited to another person by the intermediary of sound waves

[7] A phatic expression is one whose only function is to perform a social task, as opposed to conveying information.

- The sound image of this word in the brain of the second person

The written language contributes two additional links as follows:

- The sound image of this word in the brain of the first person
- The written image of the sound image in the brain of the first person
- The written word moved to another person(s) with the help of various tools as post or net
- The written image of the sound image of this word in the brain of the second person
- The sound image of the word in the brain of the second person

Both sound and written signs are used as language standards.

1. A natural language uses miscellaneous grammatical moods as indicative, imperative, conditional, injunctive and other.

The main grammatical mood is indicative and then followed by imperative, while other moods are supplementary forms based on their semantics. Thus, the conjunctive mood is a special form of the imperative mood requiring an answer from the message's receiver. The conditional grammar mood is a form of indicative used for producing conditional sentences.

2. All natural languages are *semantically equivalent*, i.e., all of them are served by the same information base, which consists of images and algorithms allocated in the brains of human beings. All relevant notions and phenomena find their expression and representation in a natural language whereby each language creates its own expression means, which however are connected to the same semantics base.
3. All natural languages are *interchangeable*. The same information can be translated from one natural language to another by persons who are in control of source and target languages. If a person knows more than one language, he/she can express the same information with miscellaneous languages. The prerequisite for translation is the presence of definitions of communicated notions in both languages.
4. All natural languages are *fragmentary*. The standard language is nothing but a normative system, whereas each person has his/her own individual level of language control. A person can use a natural language for communication even if he knows only a small part of it. The more words, idioms, grammar structures known by a communicator the more information he/she can communicate, but even 10-20 words and a couple of simple sentences can be sufficient for a communicator such as a two-year-old child.

5. *Semantics primariness*. Because non-spontaneous communications are subordinated to some purpose(s), they mainly target the transmitted meaning while the language itself can vary or even (in extreme cases) be missing altogether.

In other words, non-phatic communication is not the sovereign process but the result of the preceding task looking for ways to send/receive a particular semantic content. Thus, a person searching for a particular product in the supermarket has a lot of options, like ask the employees working at the supermarket or other customers, study the supermarket's layout, look at the indicators on the shelves, or simply walk up and down the aisles looking for the desired items.

6. Natural languages are unlimitedly *extensible*. A natural language can be extended by new notions and can be used in various communications. All existing languages were developed by adding new notions when new realities emerged in respective societies. Modern English is far more developed than during Dickens' time, because many technical appliances and kinds of social behavior were not known at that time.

On the other hand, the number of expression ways in which a particular notion can be used is fixed. For example, there are various aspects regarding shoes like buying, selling, carrying, throwing, producing, putting in, resembling and others, however, the number of all usages of this concept is ultimately limited by shoe features.

The development of society makes it possible however to extend the number of admissible usages. In earlier times, shoe soles could only be cut from something (mainly from animal skins) or spun. Now they can also be molded.

7. *Universality*. This feature of a natural language is the ultimate result of its extensibility. The complete information that is used within a society – whatever size it may be – can be expressed in its entirety in a single natural language. Basically there are three ways of expressing information:

- Direct expression of information in this language
- Translate information expressed in other languages to this language
- In case the target language does not contain the required notions, extend language by missing notions and then repeat steps one and/or two

8. *Specialization*. By their universality, natural full-fledged languages (English, German, Russian, Chinese etc.) differ from restricted jargons (Pidgins in their original form (Yule 2006), Airspeak, Seaspeak (Strevens and Weeks 1984); (("International Civil Aviation Organization" n.d.)). Jargons are used for communicating information defined in relationship to a specific activity, profession, or group. The important feature of a jargon is the restricted communication environment. The

communication occurs either on only one topic or a group of somehow related topics (Airspeak, Pidgins). Nevertheless, even a jargon has the possibility to extend like in the case of Pidgin languages extending into Creole, full-fledged languages.

In fact, all natural languages were developed from primitive-based jargons that came into existence in archaic societies

5.4. The Paradigm of Programming language

5.4.1. The Failure of Programming Language Theory

The reasons for the wrong development are rooted in early computer history. The first computers were created during the Second World War with the purpose of mathematical calculations for military needs. This development was intensified after the start of the Cold War, so it is not astounding that most programmers of that time were mathematicians and logicians who viewed programming as a branch of mathematics.

Their theoretical preferences, however, played no role in the first decade of programming. Direct programming in machine code was a very exhausting and slow process, so the programmers had to think about alternatives, which consisted in the development of programming languages. The turning point was the development of the first high level programming language FORTRAN in 1956, which for the first time allowed programs to be written oriented on its logic and not on the implementation of machine code. Other high-level programming languages followed soon after and the creation of the theory was put on the agenda.

During the subsequent years, the practitioners became theoreticians and produced a rigorous mathematized theory, which basically ignored all non-mathematical approaches supposedly as non-scientific. Often the same person contributed to both areas, as did John Backus who designed BNF and led the team that created FORTRAN – two decisive developments, which created the modern programming we know today. He also defined a new class of functional programming languages, which was completely in accordance with all the rigorous requirements of programming theory, but useless from a practical standpoint.

The fact that the theory did not cover the characteristics of practically usable programming languages was then seen as a minor problem, because these languages were viewed as a temporary solution needed for bridging the time until the maturing of the theoretically correct programming tools. Such tools were never developed, but the approach of the high-minded theoreticians remains the same also in this time as the article "Programming Language Theory" in Wikipedia convincingly demonstrates.

This article enumerates 27 of the most important works in the area, none of which is devoted to mainstream programming languages which are merely mentioned as a rather outsider phenomenon. Of the four people who have made the biggest contributions to programming: John Backus – the developer of Fortran and BNF, Nicklaus Wirth – the creator of Pascal, Dennis Ritchie - the creator of C and Bjarne Stroustrup - the creator of C++, only Backus is mentioned. The reason is the aforementioned class of function-level programming languages (Backus 1977) which he proposed in 1977.

Actually, there are only two cases where practically useful software was developed on the ideas of computer science theory. The first is the database programming language SQL, whose theoretical basis was formulated by (Codd 1972) in the year 1970. SQL, however, is a specialized language whose representation domain can be nicely described using mathematical means which is not possible for general-purpose programming languages such as C++, Java C#.

Another case is the development of compilers based on the translation theory whose development started soon after appearance of FORTRAN. The biggest problem at that time was the very slow compilation due to extremely scarce resources. Thus the IBM-1401 ("IBM 1401," n.d.) that appeared in October 1959 was sold in configuration 1400-16000 6-bit characters and its peripherals included card-readers/punches, paper tapes and drivers for magnetic tapes (monthly rental costs started at around $2,500 - about $20,987 today). The most prominent computers of the sixties IBM System-360 often possessed less than 100KB internal memory, which was neither sufficient to hold a complete compiler (a pretty big program), nor the source code with the intermediary results of compiling. Therefore, the compilers were built from many modules, each reading its input data from a hard disk or tape, processing them and writing its own file(s). Taking into account that even the smallest change to a source program required a new compilation, the latter was a genuine bottleneck consuming a considerable part of the limited and expensive time granted to the programming activities.

So theorists tried to optimize the compiling process by developing formalizations based on logic and mathematics. The set of methods including among others the formal classification of languages based on works of Noam Chomsky was created this way. The methods of compiler development based on these formalizations were complex and not intuitive but they were practically used in all compilers. Looking at this problem from the viewpoint of modern OOP, these methods were simply redundant, because the complete process of language translation could be expressed with the same effectiveness in terms of conventional C++ classes without involving whatever mathematical formalizations.

Excluding the aforementioned cases, the programming language theory occurred to be completely irrelevant. Neglecting mainstream programming went so far that the theoreticians even failed to produce a generally accepted definition of a programming language because such a definition could not be made in the conceptual coordinates of recognized theories. The lack of such a definition was thoroughly described by Jean Sammet as early as 1969 in her famous book "Programming Languages: History and Fundamentals"(Sammet 1969), but the described status quo has not changed since that time. Most modern books about programming languages simply avoid programming language definitions directly appealing to the implicit meaning of these notions (Sebesta 1993). Others, obviously guided by reasons of scientific politeness, again produce non-informative definitions like "any notation of algorithms and data-structures" (Pratt and Zelkowitz 2000).

Another proof of the incapacity of this theory is the fundamental flop, which occurred in the development of the means of formal representation of a programming language. The idea itself is nearly as old as programming languages themselves and the initial work was carried out by Noam Chomsky(Chomsky 1956) who proposed a containment hierarchy of classes of formal grammars and formulated the idea of a universal grammar.

The next step was taken by John Backus who developed the formal notation known as BNF (Backus Normal Form). BNF was first presented in the paper describing IAL, which was later renamed to ALGOL. The method, which until now remains the only practical method used for the formal representation of a programming language's syntax, was formulated by Backus as follows: "There must exist a precise description of those sequences of symbols which constitute legal IAL programs… For every legal program there must be a precise description of its 'meaning', the processor transformation which it describes, if any…" (Backus 1959).

Despite the fact that BNF failed to achieve the second part of this objective, it inspired further intensive research, which produced miscellaneous methods for formally expressing a program's meaning. The methods developed in the decades following the creation of FORTRAM are traditionally grouped into three general approaches known as denotation semantics, operational semantics and axiomatic semantics (Pratt and Zelkowitz 2000). Unfortunately, none of these approaches was ever able to attain any status outside of pure theory, because of the lack of useable results. It is very difficult to imagine that a programmer studying some programming language could extract any useful knowledge from formal specifications of this language produced with the help of known methods of semantics representations.

Taking into account the longevity and intensity of the research, its inclination towards formal mathematical models and the diversity of developed solutions, the only plausible explanation for its failure is the presence of certain fundamental constraints basically hindering the effective solution in the proposed ways.

Such a culprit exists in reality and consists of understanding the task of language specification, which is traditional for programming . The interpretation of this task shared by all known methods of semantics specification was formulated by Jean E. Sammet in the following way: "To define a language, some language must be used for writing the definitions. This latter is called a metalanguage. It is a general term, which can include any formal notation or even English itself. Metalanguage is a relative term since it is itself a language which must be defined, and that requires a metametalanguage...." (Sammet 1969).

Oddly enough, the distinction between a language and a metalanguage is relatively unknown outside of computer science. The English used to write English books is the same as that used to compose other English texts and the difference consists exclusively in the discussion's subject. According to ("What Is Meta? - Definition from WhatIs.Com" n.d.), the term "meta" is commonly used in a much more unassuming way: "*Meta* is a prefix that in most information technology usages means "an underlying definition or description." Thus, *metadata* is a definition or description of data and *metalanguage* is a definition or description of language".

In other words, the metalanguage needed for composing the specification of (e.g.) English language is a particular subset of English expressions and English phraseology best suited for producing particular descriptions, but not a special language as it is normally assumed in computer science.

This metalanguage misinterpretation effectively prevented the self-descriptiveness of the programming language, because it attributed different parts of a programming language by two incompatible systems of conceptual coordinates. The first system introduced by the data model of a programming language is utilized on the level of the coded algorithm. It uses such notions as classes, functions, instances and similar categories. The alternative perspective, based on notions of linguistics, mathematics, and logic, complemented by permanent references to the tacit knowledge is used for the expression of language syntax and semantics. For more regarding this topic, see Section 5.5.

5.4.2. The Adequate Definition of a Programming Language

Real programming is a pure construction activity that has nothing in common with mathematics. A programmer creates a program using FORTRAN, C++, PHP or any other programming language in the same way a bricklayer builds a house using a trowel, bricks and mortar. Surely, there are cases when mathematical knowledge is necessary like operations with graphics whose transformations are represented by mathematical means. Generally speaking, a programmer does not need any more mathematics than a good ninth grade student does and, following the same logic, the programming language itself, also does not need any mathematical constructs for its adequate definition.

This understanding of programming was, however, absolutely inacceptable for the highly ideologized theoreticians of the programming language, because in their eyes, the programming was just another branch of mathematics. For this reason, all programming tools built using pragmatic concepts were seen as unscientific and so they chose to reject the known adequate definition of a programming language because of its non-mathematical background.

Actually, the definition given by Saul Gorn in a short article in the year 1959 (Gorn 1959) is one of the oldest definitions of a programming language ever. Here is the oldest formulation to be found in the ACM library: “The point of view expressed in this paper makes more tangible two principles accepted intuitively by many programmers and logical designers. They are: a) the equivalence of formal languages and machines, b) the equivalence of programming and hardware.”

The same idea was expressed in 1978 by Yaohan Chu (Chu 1978) “To each programming language, there is associated an ideal computer architecture which executes the program written in this language. This ideal architecture images the control constructs and the data primitives of the programming language. It is a virtual architecture, because it may not be possible to be fully implemented by real hardware architecture. If the programming language is a high-level, the virtual architecture is a high-level architecture.”

Because I was not aware of the definitions above at the time of writing this paper (Sunik 2003), I used my own definition, which is basically the same: “A programming language is essentially the machine code of the processor implicitly introduced by the definition of this language.”

The commonality between these definitions is the interpretation of a programming language as a combination of two things differing by their nature: a **genuine language**, which is a system of linguistic signs used in some communication and a **processing unit** executing programs composed in this language.

A processing unit is a pure logical construct that is not intended for whatever physical implementation. Instead, a source program is either directly executed by a software interpreter running on a physical low-level processor, or is translated to the machine code of a physical processor with the help of a compiler.

A genuine language is a linguistic tool of limited usage. Its purpose consists of instructing a processor and its representation world is restricted to the depiction of bit sequences allocated in the computer memory and the sequentially organized manipulations with these bits.

In this paper, a logical processor implicitly given in the language definition is designated as an *innate processor* and a computer powered by this processor as an *innate computer*. The functionality of an innate processor can be completely emulated by a plain non-optimized language interpreter running on the top of a real low-level processor. An innate computer can be simulated with a real computer executing source programs with the intermediary of such an interpreter.

For the purpose of this article, the difference between a real and an innate computer can be reduced to the differences in the executable files. Assuming that both computers have an identical application `HelloWord`, in order to run this program on a real computer, a user has to compile the file `HelloWorld.cpp` into the object file `HelloWorld.obj`, link the latter getting the executable file `HelloWord.exe` and start running the executable. Because the processor of an innate computer understands C++ code, a user of such a computer simply starts `HelloWorld.cpp` without any additional preparation.

The relationship between a genuine language and its processing unit processor are essentially the same as the relationship between a low-level machine code and a physical processor running it. For example, the low level i386 machine code is run by processors of the i386 family. Computer processors of other architectures are controlled by codes composed in their own machine languages. Accordingly, an innate processor of C++ is controlled by C++ control structures, while an innate processor of Lisp is controlled by Lisp expressions.

The innate processors of different programming languages possess distinct architectures, execute distinct commands and run their programs in distinct ways. Differing from simple instruction types of low-level processors, innate processors of high level languages support hierarchically built instructions.

An innate processor normally includes the main processor directly executing source code of a programming language and a low-level coprocessor executing low-level (assembler) functions.

The following describes the structure and working principles of the innate processor of C++

5.4.3. The Structure of C++ Innate Processor

5.4.3.1. Storage

The internal storage of an innate processor of C++ is a sequence of bytes used for the allocation of the code storage containing the running program and the data storage. The latter includes the following components:

- public and static objects
- the program's stack with the automated variables
- data allocated with `malloc` routine and released with `free` routine
- data containing the identification tables of the called functions and the allocated automated variables
- identification table of program's global objects (publics)
- array of module identification tables, each describing the content of the respective module
- working storage used for the recognizing and processing the current command

Other storage components are internal registers like instruction pointer (IP) and others.

5.4.3.2. Command Set

- Control structures (*if*, *while*, *for*, *do*, *switch*, *case*, *break*, *return*, *continue*, *goto*)
- declarations and definitions
- the function call statement and the method *execute* actually executing the called function
- operators with built-in fundamental types (as `int operator=(int op1,int op2);...`).

5.4.3.3. Data

A processor manipulates instances of built-in fundamental types *char*, *short*, *int*, *float* and user-defined types (*structs*). The latter are sequences of instances of user-defined and fundamental types.

Fundamental types are built-in classes with operator functions as their methods. Their general superclass is a byte sequence allocated in the work storage. Its attributes are the address and the size in bytes. Methods of fundamental types are operator functions processing type instances.

5.4.3.4. Code Structures

- A program in C++ is a sequence of modules (files), which are read from the external memory into the code storage when starting the program.
- A module is a sequence of declarations and definitions that defines a unique (non-typed), unnamed instance.
- Declarations introduce classes, functions and externals. The externals are instances of fundamental and user-defined types defined elsewhere.

- Definitions represent instances (global and static) of fundamental and user-defined types actually allocated in this module.
- The communication between modules is effected by means of declaration of externals, actually defined in one of the present modules.

5.4.3.5. Command Execution Order

Assuming a C++ program is semantically and syntactically correct, its execution includes two steps.

The initialization step starts from reading the complete program modules, storing them in the code storage and generating the identification table of public instances as well as the array of module identification tables.

After that, an innate processor consequentially processes each module. For each declaration, it creates an entry in the identification table of the respective module. For each instance definition, it invokes the constructor creating a new instance. For each non-static declaration, it also creates an entry in the public table.

External declarations are stored in the temporary table for subsequent processing.

After reading all modules, an innate processor walks the temporary table of external declarations and replaces their occurrences with respective declarations/definitions from the public table.

The execution step begins from the call of the function *main*.

An innate processor reads commands, recognizes their types and passes them to the respective decoders. The method *execute*, invoked by each call of a C++ function, carries out ALU functionality when operations have to be performed with fundamental types.

Each object instance is characterized by a number of attributes such as the identifier (for a named object), the type name, and the position (absolute addresses for globals, offsets for locals and `struct` members, etc.).

Calls to external low-level functions invoke the low-level coprocessor.

5.4.3.6. Implementation

An innate processor of C++ can be implemented by three possible ways, of which only the last is used practically:

1. Creating a physical device executing C++ code
2. A logical device emulated by a non-optimized software interpreter running on the low-level processor. It is an application composed in the machine code, which completely

reproduces the innate processor's logic. The input parameter of an interpreter is a program composed in this programming language.

3. With a low level processor running a low level program that was produced by a C++ compiler and a linker. A compiler is a code-inliner that, on the basis of a source program and its own built-in knowledge of the C++ interpreter, produces a degenerate version of an interpreter reduced to the execution of this single program. The compiler pre-executes the interpreter, optimizing away all operations with constant objects, and generates the optimized code of all calls.

5.4.3.7. C++ Computer

A *C++ computer* is an imaginary PC controlled by an innate processor of C++. This computer is functionally equivalent to a conventional PC because both computers produce the same results in the same way while executing the same C++ program.

Principles of operations and organization of a C++ computer are also similar to those of a conventional PC. It has a keyboard, a mouse, a monitor and is supervised by some OS. Its programs are produced by a programmer who uses some editor to input the program into the computer. After preparing a program, a programmer stores it in a file, after that it can be invoked in this or another way (e.g. from the command line).

A communication process between a programmer and a C++ computer includes several communication chains produced during various operations with code. Among these operations are:

- Sending the input into a program. A programmer inputs program into a C++ computer with the help of keyboard and some editor program. During the process of producing a program, a programmer can see its current state on the computer's monitor.
- Storing a program in permanent storage (hard disk, flash, etc.)
- Program start up. A program is started by a user of a computer (e.g.) from the command line. For that, it has to be loaded from the permanent storage into the binary computer's memory.

Direct communication between a programmer and a computer contains communication chains with a minimum of three elements: a) items allocated in the programmer's brain; b) identifier of keyboard key allocated in the programmer's brain, used during typing process; c) items allocated in the binary memory of a C++ innate processor.

Reverse communication between a computer and a programmer is used to convey the program already found in computer storage to the programmer. This communication chain also includes three codes: a) items allocated in the binary memory of a C++ innate processor; b) external items consisting of alphabetic characters placed on either computer monitor or printed on a sheet; c) items allocated in the programmer's brain.

The communication chain used in read/write processes includes two codes - the binary number used in the binary computer memory and the code is specific for a particular information carrier. For example, hard disks use codes consisting of magnetic particles allocated on their surfaces etc.

5.5. The Paradigm of Universal Knowledge Representation Language

5.5.1. The Failure of Knowledge Representation

The discipline of knowledge representation (KR) studies the formalization of knowledge and its processing within machines. It was established as a separate branch of AI during the 1970s when the first tasks related to intellectual reasoning came to the agenda. The methods of KR include production rules (if ...then ... statements), semantic networks, frames, first order logic. Of all these methods, only rule-based knowledge representation was ever used for practically relevant developments. During the eighties, it was applied in a number of large-scale projects, and found its way into many industrial enterprises and government applications (Hoekstra 2009).

None of the other KR methods was ever used for industrial programming, because the mainstream programming languages (C/C++, Java etc.) were essentially better in respective implementations. At this time, the development of intellectually loaded applications did not differ from programming of complex "non-intelligent" systems, as both areas require building millions of lines of code running on complex hardware systems.

Actually, we need to speak about two completely different KRs – those grounded in scientifically proven principles but failing in practical developments and those based on the concepts of mainstream programming and demonstrating efficiency unachievable by the former. The only concept shared by both KRs is object-orientation.

Similar to the programming language theory, KR is also poisoned with the penchant for the scientific pureness. While several decades were spent on the intensive study of scientifically-proven even if practically inapplicable methods, the KR-relevant characteristics of mainstream programming languages were stubbornly ignored because the methods of these languages were supposedly "unscientific". This is the second case this term is used for characterization of the mainstream programming and this is not an accidental coincidence.

The unscientific methods are not discussed in the scientific literature, so no references can be provided[8]. However, it is exactly this characteristic of mainstream programming languages that I get as an answer each time I question specialists regarding why they do not study the KR features of these languages. It is clear that formal languages with such expressive power and effectiveness cannot be characterized in this way, unless it is an excuse for suppressing undesirable discussions.

In fact, such a discussion is merely impossible, because the conceptual world of KR simply lacks the necessary concepts. This theory was created and developed as a branch of theoretical computer science and its understanding of programming languages is borrowed from another branch of the latter, known as the "the theory of programming language". Even though its name suggests otherwise, this theory however never studied the programming languages actually used, concentrating instead on the mathematical and logical basis of programming, which is something that does not exist in reality (see Section 5.4.1 for details).

In other words, KR theoreticians have no other choice than label the mainstream programming languages as unscientific, because this science uses the term "programming language" for designation of something that not exists in the real world.

****

The obvious mistake of the traditional approach is the total disregard of the quite obvious circumstance that the effective KR can be built exclusively on the basis of a *universal knowledge representation language*. Most of the knowledge we have is represented namely in such a language[9]. Each natural language is an ultimate KR tool, which can be used by everyone from infants learning their first words to scientists discussing the most complex theories. The only (however grave) problem is that modern computers have difficulty understanding texts composed in a natural language because these texts suffer from ambiguity and a vast amount of implicit background information.

This problem, however, can be solved with the help of an *unambiguous* language. Curiously, such a solution has never been seriously discussed in the scientific community. The reason is the widespread belief that the universality (expressive power) and unambiguousness are supposedly non-combinable characteristics. See, for example Sowa: "The expressive power

[8] There are not many works discussing the general basics of KR, the most prominent of them is the classical book of Sowa (Sowa 2000). Among other examples, discussing general approaches, are the thesis on the history of KR made by (Hoekstra 2009)and (Davis, Shrobe, and Szolovits 1993). None of them ever mentions the theme.

[9] In general, there are three ways of gaining knowledge: from general experience (tacit knowledge); from sensorial images ("one look is worth a thousand words") and from communicating in a natural language. The last way is the most powerful, but logically it depends on the first two, which provide the basic information. More on the theme in (Hoekstra 2009).

of natural languages, which is their greatest strength, is also one of the greatest obstacles to efficient computable operations" " (Sowa 2000).

A search for any kind of explicit proof supporting this negative stance however doesn't turn up anything, except for a few banalities about the primitiveness of computers and the complexity of human knowledge in general and human languages in particular. Contrary to general belief, this opinion is not based on the particular results of any thorough language study, but from the lack thereof or to be more exact from the discrepancy between the study of the natural and the programming languages, which came into existence at the very beginning of computer science.

In actual fact, the belief regarding the universality and the unambiguousness as mutually exclusive characteristics is the main reason why the development of the universal formal language was never put on the actual agenda of the scientific community. Fortunately, this view is very far from reality. An unambiguous language can be just as universal as a natural language is, for that however, it has to be extended by several features basically missing in programing languages.

5.5.2. *Definition of Universal Knowledge Representation Language*

The reason why creators of programming languages never showed any interest in their KR capabilities, lies in the self-sustainability of programming whose target domain remained the same since the time of the first computers. The main participant in the programming process is a programmer who gains knowledge with the help of natural languages, in the form of various images or as tacit knowledge. A programmer uses acquired knowledge for issuing unambiguous instructions to an innate processor.

In the case where all complex knowledge-related activities are effectively performed by humans, there is absolutely no stimulus for extending a programming language to the tasks lying outside of its original purpose. One of consequences of this approach is the paradoxical situation in which a language used for producing formal representations can only be described with the help of informal categories. Concretely, while the meaning of the character literal 'a' is unambiguously defined in the language's definition, the meaning of the letter 'a' in the language alphabet is completely undefined and hence misunderstood.

The adequate definition of a programming language solves this paradox by revealing the missing links between the real world and the binary entities represented by the language code. As shown in Section 5.4.2, a programming language is essentially a specialized jargon applied in the single monologue discourse occurring between a programmer and a processing unit. A programming language basically disallows composing any other texts except programs executed by this processing unit.

The communication in a programming language occurs between a programmer and an innate computer. A programmer studies a language by reading the definition of this language (let us assume the definition is given in a readable format) and applies it by composing a program, which he sends to an innate computer, in turn executing the commands. In OO terms Figure 5-2, this communication environment includes an object `Programmer` generating a `Program`; an object `PLComputer`, executing the `Program`; an object `Book`, from which a `Programmer` studies the language; a class `Language`, which is the multilevel structure based on alphabetical characters used in the communication exchange between a `Programmer` and a `PLComputer`.

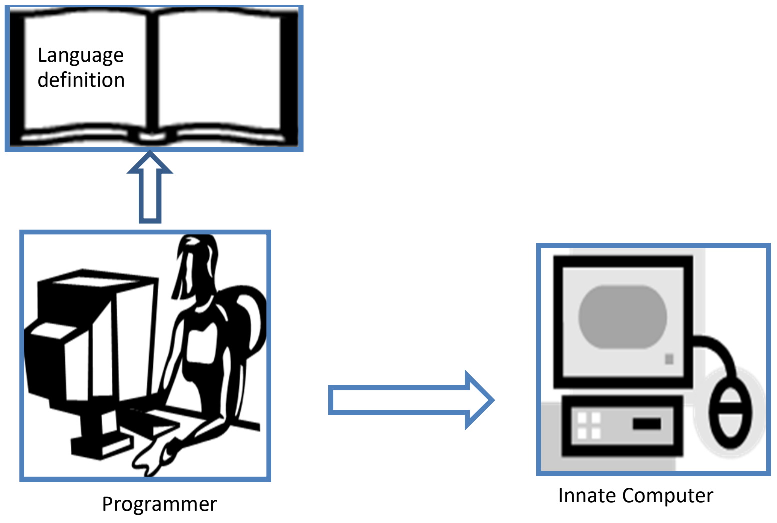

Figure 5-2

Even though the system of programming language signs is dedicated to the representation of a tiny fragment of the real world, similar to any other jargon it is unrestrictedly extensible. Taking into account that all currently existing full-fledged natural languages were built from the primitive jargons used by ancient societies, *every programming language can be considered as the embryo of the respective universal representation language*.

The difference between a programming language and its full-fledged extension is that the latter is not restricted by whatever predefined communicators or communicated themes and can be used for expressing any information ever formulated in any human or computer language. Such a language can be built by extending the linguistic signs of a programing language by expression abilities needed to represent entities exceeding the representation world of this programming language. This language is not restricted by whatever predefined communicators or communicated themes and can be used for expressing any information ever formulated in any human or computer language.

Theoretically, the world of full-fledged formal languages can include many of them, but in reality, such plurality is rather disadvantageous, because all these languages will ultimately possess exactly the same expression power. Different programming languages are also very different in their expansion abilities. Some programming languages are more extensible than others, with the best match being the maximally extensible of them - C++. Because this language delivers the most prominent example of language extensibility, it has also been chosen as the definitional basis of the full-fledged representation language ***T*** considered in the next sections.

Structurally, ***T*** is nothing more than C++ extended by several features. The most important of these is the mechanism of grammar moods, allowing non-executable representations to be compose and makes the language self-descriptive.

The main grammar mood is indicative and used to compose narrative (non-executable) texts. Another essential grammar mood is imperative that is used to compose commands executed by a receiver of imperative texts. If necessary, the language also allows conditional, subjunctive and other grammar moods, but their creation and use is not considered in this book.

The narratives are the dominating code-form of natural languages. A narrative text is a sequence of data items depicting miscellaneous entities (objects, events and pluralities) and their features. Because narratives are not associated with the immediate processing of referred items, they allow the representation of concepts, objects and algorithms existing outside the world of bit sequences allocated in the computer storage.

The self-descriptiveness means that the structure and the semantics of a ***T*** discourse can be described by its own code. This is normal for a natural language but, despite miscellaneous attempts, this feature has never been achieved in programming languages. Section 5.4.1 describes reasons, which have historically prevented the self-descriptiveness of programming languages. An example of a self-descriptive specification is given in 6.20.

5.5.3. The Logic of Communication Exchange

The basic code-form of a universal representation language is a narrative text consisting of indicative sentences. The narratives create the basis for other language applications. The model of narrative communication can be demonstrated on an example of the simplest non-executable language allowing composition of a single sentence stored in a variable of the type bool. The language only has the two values `true` and `false` produced by the comparison procedure `isEqual`.

```
int  num1, num2;
bool isEqual(int, int);
book bVar = false;
…
bVar = isEqual(num1, num2);
…
if( bVar == true ) foo();
```

The output value produced by `isEqual` designates a meaning. The sentence "`true`" means "`num1` and `num2` are equal", the sentence "`false`" means "`num1` and `num2` are not equal". The unambiguous representation of the meaning requires linking with its comparison procedure. See a grammatical construct implementing this functionality in 6.14.

A text in `bVar` is a copy of the original value produced by `isEqual`. While the original sentence is an assertion proving the fact of equality, a copy is an assumption prone to possible distortions, because it can also receive a value in a result of a simple assignment "`bVar = true`" without any invocations of `isEqual`.

The semantics of the text in `bVar` includes the semantics of the code writer (`isEqual)` but can also include the semantics of the complete algorithm, if `isEqual` is logically connected to other code parts. Additionally, it can include the semantics of a code reader in case the comparison procedure `isEqual` used by the latter is different from `isEqual` from the code writer.

A comparison procedure is the simplest **observer**. It compares an arbitrary number of entities, returning the value designating the comparison result. The complex meaning is the collection of simple and complex meanings. Complex meaning requires complex observers including mulitple comparison procedures.
An observer is an object, which is able to execute distinct comparison procedures and use it for this or other purposes. The crucial feature of an observer is an ability to send/receive a text with the comparison result to/from another observer. The purpose of information exchange consists of providing the alternative way of getting the results from comparison procedures, which allows collecting information from remote environments. The exchange can only take place when both a writer and a reader are equipped with the same comparison procedures and already possess the necessary knowledge regarding compared entities.

An observer can receive information in two ways. The primary method of information acquisition consists of a direct comparison of the required entities. This method however can only function if an observer has immediate access for respective entities and can actually perform all required comparison algorithms. Otherwise, it may receive information in the form of a text in some language produced by another observer.

Assume that there are two text exchanging objects named `firstObject` and `anotherObject`, which implement the functionality from the above example. Each object can get the result of comparison by either executing `isEqual` or querying its counterpart. Thus, `firstObject` can get the result by invoking the function `getBoolVal`, which is the part of the communication interface

```
bVar = anotherObject.getBoolVal("isEqual(num1, num2)");
```

If both communicators refer to the same num1 and num2, this mechanism proposes the alternative way of getting the same knowledge. This can be useful for getting already acquired knowledge without spending its own time executing the comparison algorithms.

In all other cases, when there is no guarantee that both observers have access to the same `num1` and `num2`, there is also no guarantee that a delivered value actually reflects the result of the comparison process. Thus, the statement producer can produce this statement by getting it from the third observer, by returning the result of comparison without actual execution of `isEqual` or by reusing some obsolete value once produced in the past.

These observers who do not possess the function `isEqual` are also not able to understand its meaning.

Imperative communication is another form of communication exchange. It consists of sending instructions to an object which is able to recognize and execute them. Assume that `anotherObject` object supports the commanding function `doCommandWithBoolReturn`, so `firstObject` can get the result of comparison by running the following code:

```
bVar = anotherObjecxt.doCommandWithBoolReturn("isEqual(num1, num2)");
```

The function will deliver the proper result if `anotherObject` is actually able to execute the function `isEqual`. In general, a function executor does not need to understand the semantics of its instruction, but rather only know how to carry out this command.

The representation of human knowledge is completely based on the above approach. A human being is considered to be an ultimate observer, that collects information with the help of its senses and exchanges this information with other human beings in a human language. The model HI describing the information capabilities of a human is defined in 3.2.3. A Hi can generate arbitrary text message with arbitrary information. A Hi reading previously generated text will accept only the part of information, which it can understand. The crucial part of understanding is the ability to identify the text information with output of particular comparison algorithms. The misunderstandings between persons of different age, culture or education result from the inability of the reader to interpret the read information correctly.

Regarding human knowledge, there are no differences between its representation and the above example except the complexity of the former. Thus, the indicative phrase "Pete puts the cup on the table" designates the process, participants of which (Pete, cup, table) are complex images, each including a large number of visual, sound and tactile components.

Below is the characterization of the indicative variant:

- The sentence describes putting a particular cup on a particular table. The brain of an HI receiving this sentence, recognizes the associated meaning, which is a certain internal representation (image) stored in the internal memory.
- It is implied that a reader already has the respective internal image, which includes multiple values produced by the brain’s comparison procedures. The image was formed in the past when the reader’s senses registered similar event(s) occurred in its vicinity.
- The image does not represent a concrete event but its type. A concrete event (or events) complying with this image can occur everywhere, anytime or even be imaginable.

The semantics of the imperative sentence “Pete, put the cup on the table" is as follows:

- The sentence is a command, which its producer gives to its reader Pete.
- The designated process can only occur if its reader is able to take the required cup and put it on the required table.
- The designated process will only occur if its reader actually executes the command.
- A reader may also execute the command without understanding the semantics of the sentence, e.g., when the code reader is a robot and this command is one it can carry out.

To summarize, a reader of an indicative code has to be able to understand the meaning of the message, but does not need to have a physical connection to the action's environment. Those, who do not understand for this reason, are excluded from the narrative communication. In the case of imperative communication, a code reader has to be able to produce the required actions but does not necessary need to understand the command’s meaning.

6. The Universal Representation Language *T*

6.1. General Characteristics

T is an artificial human language based on extension of notational and grammatical system of C++. The expression means of ***T*** contains practically all C++ concepts including real, abstract and virtual classes, instances, procedures, multiple inheritance, templates, control statements (called sentences) etc. It has also several new concepts, which are necessary for non-executable representations. Because of new concepts and irregularities of C++ syntax, the syntax of ***T*** is only somewhat similar to that of C++.

T is purposed for information exchange between various communicators. The communicators are either physical objects (human being, devices, computer programs) or logical entities, which are (or considered to be) able to read and/or write code composed in this language and execute designated actions. A human being is the ultimate communicator possessing all language functionality, while other communicators could possess limited abilities.

T is not restricted to a particular representation domain or discourse and allows composition as executable as non-executable code from which the latter is the default code-form. It exploits the TMI view of the world, according to which, every real or imaginary entity can be viewed as an object whose behavior can be expressed in the manner of programming algorithms. Texts composed in various natural languages can be unrestrictedly translated into this language.

T expresses conventional knowledge about objects, actions, properties, states with the help of miscellaneous sentence types as declarations, definitions, occurrences of procedures, conditional structures *if*, *for*, *switch* etc. Names of types and instances stay for nouns, verbs are represented by procedure names, the language also allows adjectives adverbs, pronouns, prepositions, conjunctions and quantifiers.

The lexical content of ***T*** is defined in dictionaries, which contain declarations and definitions of miscellaneous types, functions and instances. Assuming there is sufficient vocabulary, each text composed in any of human languages can be adequately translated in ***T*** in the same way as it can be translated to any other human language. Furthermore, the language allows explicit expression of kinds of information traditionally considered as inexpressible like tacit knowledge or even non-human knowledge.

T allows multiple implementations. An implementation of ***T*** is its application in some communication environment where the whole language or some subset of it is used for the information exchange. The language allows the creation of a monolingual communication environment in which all programming tools existing on the system will be explicitly considered as miscellaneous communicating objects commanded by programs composed in this language. Additionally it can be used for writing various non-executable specifications (vocabularies, documentation etc.)

T is its own meta-language using the same expression means for both the specification of a communication environment and for composing discourses exchanged in this environment. The ability of a language to specify its own grammar and semantics (meta-representation) is essentially similar to that of natural languages (English, Russian, Chinese etc.). See section 6.20 for additional information for meta-descriptions.

The formality in ***T*** is understood as the compliance between the formal and actual parameters of the respective procedures. The check for the compliance between the formal and actual parameters is the only action generally defined for the processing of ***T*** code. This is completely sufficient providing all entities referred with the help of this language are described in the terms of OO categories.

6.2. Expression Power

The expression power of ***T*** can be shown using the following C++ example:

```
int i1 = 5;
Complex c1 ( 3.0 , 4.0 ); //a complex number consisting of two real numbers
```

The C++ meaning of this code sums up the semantics originating from four different sources.

1) Language documentation. The object `i1` is an instance of a fundamental type `int` whose description can be found in the definition of C++, composed in a natural language, normally English.

2) C++ code. The object `c1` is an instance of a user-defined type C`omplex` whose complete definition is made in C++.

3) Foreign knowledge. While the names `int` and `Complex` designate binary objects, their semantics are essentially extended by the associations with the mathematical theory defining integer and complex numbers. Similar to the C++ documentation, the description of the mathematical theory is also made in a natural language.

4) Comments. The text after "`//`" gives additional information in English.

The code above can also be interpreted as a ***T*** code completely emulating C++. Such a reinterpretation is semantically indifferent but it radically changes the associated lexical environment, because all notions, which in the case of C++ are made in a natural language, can now be made in ***T***. In contrast to C++, ***T*** allows all aforementioned specifications including the complete specification of the C++ innate processor, the mathematical content and comments.

6.3. Communications at first glance.

An application can include many modules. A module is a text composed in ***T***, which includes a sequence of declarations, definitions, labels and commands devoted to a particular subject. A module can be a paper book, a book's part; a file in the computer storage, a file's part.

A module can include narrative and executive information. Narrative information (*info*) is essentially a value or a collection of values describing something. Executable information (*rule*) requires execution of actions of some form from a module's reader.

Communicators can be referred to in ***T*** code with the help of predefined names `_reader` and `_writer`. Events of reading and writing are designated as respectively `_read`, `_write`. The features of communicators can be completely described in code and such description, if present, constitute the basis of code semantics. The differences between C++ and ***T*** can be shown using the famous "Hello World" example:

```
void main()
{
    printf("Hello World");
}
```

A ***T***-application with the same semantics includes the same code extended by the specification of communication environment. In this case, only a reader of this text has to be provided:

```
_reader := _cplusplus              //the definition of the code reader
proc main()
{
    printf("Hello World");
}
```

Similar code can be used in completely different communications. Assume there is a logical processor called `cplusplus_logger`, which registers operations performed by an innate processor of C++ and writes a log file. A log produced during execution of the aforementioned example can be grammatically similar to the latter but it is a narrative enumeration of already executed functions.

```
_writer := cplusplus_logger
main()
{
    printf("Hello World");
}
```

The differences between imperative and indicative code can be demonstrated in the example of the procedure `foo` defined as

```
proc foo()
{
    foo1();
    foo2();
}
```

In ***T***, the text "`foo()`" means not the invocation of `foo` as it normally is in the case of programming languages, but the declaration of an occurrence (event) of `foo`. An event is an instance of a procedure and can also be declared in the same way as an instance of a class. The following code shows a sequence of two `foo` events.

```
{
    foo ff();                    //an event of foo named ff
    foo();                       //an unnamed event of foo
}
```

If an occurrence of `foo1()` in `foo()` was named as `f1` it could be referenced as `ff.f1`.

The command requiring execution of `foo` has to be written as.

```
#foo();                          // "#" designates a command,  T has no preprocessor in the C style
```

A command's executer will read the command body, interpret each procedure occurrence as a command and execute it. A name of an occurrence is interpreted as a command's name.

The object `_cplusplus` executes code in exactly this way. The procedure, which invokes a C++ innate processor, finds the file `HelloWorld.cpp` and invokes an instance of `_cplusplus` with this file as a parameter. An instance of `_cplusplus` looks for the procedure `main()` and executes the built-in command `#main()`, which starts reading the function's body and executing its content.

The details of non-executable code are given in the following sections.

6.4. Syntax specifics

6.4.1. Two Character Sets for Composing Words

T words may be composed from two different character sets. The conventional words consist of letters, digits, and the sign "_". The special words are formed from characters traditionally utilized for operators ("+", "-", "*", "/").

Both kinds of words possess the same rights. The name may include characters taken from only one alphabet. Hence, names composed from different alphabets may be distinguished without additional separators. In the expression "`a=b+c`" , object names (that is, `a`, `b`, `c`) are automatically separated from function names (that is, "`=`", "`+`").

`proc ---(int);`   //this is the declaration of the procedure with the name "`---`"

6.4.2. Universal Declaration Syntax

All objects including procedures and arrays have explicit types. Thus, the "C" function declared as "`int foo(int i)`" in ***T*** has to be expressed as

```
cfunc foo(int i,out int);
```

where the type "`cfunc`" is the class of C++ functions.

The declarations of the "C" array e.g., "`int ii[3]`" are changed to "`carray ii int[3]`" where `carray` is the class of "C" arrays.

6.4.3. Dot Statement

The dot operator "." allows referencing to parts of arbitrary containers. Thus, the expression `a.b` is a correct reference for the following components:

- procedure parameter: `proc a(int b);`
- template parameter: `proc a<typename b>(b c);` //b is just additional parameter
- class component: `class a {class b;};`
- namespace entitiy: `namespace a{int b;};`
- component of complex enumerations.

6.4.4. Syntax of classes and procedures

The syntax of a class declaration is `class <classname> <classtype>` where `classtype` is either a C++-like structure or a simple type name. The latter case is equivalent to the `typedef` declaration.

```
class x {int xx;};          //the classical declaration
class x int;                //same as "typedef int x"
```

The structure of a procedure is an ordering construct of any type.

```
proc prc(int k) if (k < 0) return -k else return k * 2;
proc prc_syn(int k) prc(k);
```

The last declaration describes `prc_syn` as a synonym of `prc`.

6.4.5. No keyword "template"

```
template <class cl>cl f (cl obj){return obj*2;}       // C++ declaration
proc f<class cl> (cl  obj,cl outobj){return obj*2;}   //T declaration
```

6.5. Text structure

The upper text entity is a text module, which may include following types of definitions and declarations.

- Classes;
- Procedures.
- Functions.
- Rules - sequences of commands syntactically equivalent to procedures.
- Definition of instances (classes, sets, sequences, events).

Declarations of aforementioned entities. A declaration of an entity is a reference to the proper definition of this entity.

6.6. Types and instances

Instances of a class are individual objects. The declaration "`object obj`" declares instance of any class

Instances of a procedure are individual events. The declaration "`event ev`" declares instance of any procedure

The term *instance* designates either object or event. The declaration "`instance inst`" declared instance of either a class or a procedure.

The term *type* designates either a class (type of a thing) or a procedure (type of a process).

The term *entity* designates either an instance or a type. The declaration "`entity ent`" declared anything.

- A type is referred by the type name.
- An instance is referred by either a type name or an instance identifier.

6.7. Causal relations

A causal relation is a sequence of two events, first of which initiates the second.

```
proc effect();
proc cause (out event);
cause()=>effect();
```

6.8. Grammar Moods (executable and non-executable code)

Assume there is two procedures

```
proc proc1();
proc person::proc2();
```

The procedure `proc1()` occurred somewhere and the information about its occurrence has to be delivered to the code reader. The procedure `proc2()` designates the kind of actions, which a code reader can do by request. The following code contains both non-executable and executable information.

```
proc1();                          //informs a code reader about the occurrence of proc1()
Bob::proc2();                     //informs a code reader that proc2() is executed by Bob
#proc2();                         //requires from a code reader to execute proc2()
```

A sequence of commands can be defined as a rule. The rule `rule1()` includes two consequentially executed commands followed by the narrative describing the semantics of the second command

```
rule  rule1()
{
   proc1();                       //command requiring executing proc1()
   proc2();                       //command requiring executing proc2()
   ^execute proc2()=>proc3();     //informs that execution of proc2() causes occurrence of proc3()
}
//non-executable context
rule1();                          // a command because rules are executable by default
#rule1();                         // the same as before
^rule1();                         //informs about execution of rule1()
```

6.9. Names and References

A name is an identifier permanently assigned to an entity while a reference is its temporary identifier. ***T*** allows reassignments of the same reference. A reference assignment occurs with the help of the operator "`:=`" or "`()`". In the below code, the reference `bref` firstly designates `first` and then `second`

```
int first,
    second;

int &bref := first;
bref := second;
bref(second);                //the same as before
```

The reference assignment also has a mirrored variant "`=:`" which references the left side object.

```
second =: bref;              //the same as before
```

When a reference is initialized in the declaration point, it can be defined implicitly without the character "`&`". References defined in this way cannot be reassigned again. The following definition introduce the reference which cannot be changed

```
int constRref := second;
constRfef := first;          //Wrong
```

Parameters are references by default.

A number of predefined references have special meaning.

References `you` and `I` are set implicitly. They designate the text creator and the current text reader. The same functionality has the references `_reader` and `_writer`.

References `_write`, `_read` designate events of writing and reading the current text.

A reference `that(type)` designates the last event/instance of `type`. For example:

```
class Man    :Human;
class Woman :Human;
Man Alex;
Woman Ann;
Man Peter;

that(Woman)              //designates Ann;
that(Man)                //designates Peter;
that(Human)                //designates Alex;

Ann::work();
Alex::work();

that(Human::work)        //designates Alex;
```

6.10. Unnamed entities and homonyms

Differently from programming languages, natural languages use proper names fairly rarely. The general way to refer to entities in a natural language is a reference to the type of thing, but not to its name (e.g., "a *man* near to me", "a *letter* lies on the *table*"). ***T*** allows both reference to things only with type and to refer to them with the help of an identifier and a type.

```
Hand left,right;
Leg left, right;
left:Hand;
right:Leg;
```

The unnamed object can be declared and referred to as follows.

```
Letter << >>;               //this is the declaration of the unnamed object.
:Letter;                    // this is the reference to the unnamed object of the type Letter.
proc <<>> (int pk1);        //this is the declaration of unnamed procedure
proc <<prc>>(int pk2);      //the same as previously; the name in double angles is a comment
pk2=2;                      //OK, pk can be referred to directly because of the lack of procedure name
```

6.11. Embedded Declarations

T requires explicit declaration of all user-defined entities before their first use. The only exclusion are unnamed objects defined in the list of actual parameters. The language allows embedded declaration of external entities enclosed in the dollar "$" signs. The code "`$Type$Instance`" means both the declaration as "`Type Instance`" and its immediate use. Non-executable code also allows declarations as "`$Type$`" meaning some unnamed instance of the type `Type` and "`$$`" meaning some unnamed instance of some arbitrary type.

```
class Buffer{...};
proc foo(Buffer);

Buffer buf_internal;          //the definition of the object of the type Buffer
foo(buf_internal);

foo($Buffer$buf_internal);  //the compact form of the two above lines
```

6.12. Type Specifiers

6.12.1. Heap (plurality)

T uses the concept of a heap, which is an unorganized plurality of entities. The following is the declaration of the heap `hp` of `int` binary numbers

```
heap<int> hp;
int.. hp;                                   //alternative declaration
```

Heaps are the simplest pluralities. Thus, the phrase “the flowers blossomed” describes a heap of flowers that blossom. Heaps allow a specification of relations between pluralities. The phrase “several village inhabitants” designates a small sub-plurality of the complete village population.

6.12.2. Struct

T `struct` represents a sequence of components, which can be either complex, or simple components. Consider the following structure

```
struct strt{int m_int;char m_char;};
```

This is a sequence consisting of `m_int` and `m_char`, where the first component `m_int` is a sequence of 4-byte elements. The simplest base of all components of `strt` is a char (a byte) are the complete length of this sequence is 5 bytes. ***T*** allows explicit definition of the simplest base as

```
struct bitstruct  (byte) {int b_int; char b_ch;}          //the same as before
struct bitstruct  (bit) {int b_int; char b_ch;}           //this is the struct of 40 bits length
struct treestruct (tree){array trees tree[20]; tree tr;}//the sequence of 21 trees
```

A struct can be defined as an overlay of already defined sequence (see 6.16 for details).

```
array memory byte[0x1000];
memory::struct sbytes (byte ){int obj1; array ari byte[10];};
sbytes  st;             //the structure st is allocated somewhere in memory
```

The value of the type `struct` can be given either in a traditional C notation or in the special notation of ***T***.

```
struct person {string name, Date borndate, int ssn;};
person john = {“John”, “23/03/1987”,12345678};      //traditional notation
person john = {“John” “23/03/1987”  12345678};      //T special notation no commas in a sentence
person[2] couple = {“John” “23/03/1987” 12345; ”Mary” “13/12/1988” 3238388892;};
```

6.12.3. Union

In a union, at most one of the members can be present at any time. A union is a superclass of all its components. Thus, the union `un` is a superclass of classes: `int`, `float`, `date`:

```
union un {int a;float b; date c;}
proc f(un U);
float fl = 2.0;
int   ii = 1;
f(fl);                                    //correct, "fl" derives "u";
f(ii);                                    //correct, 13 is an int constant.
```

Accordingly, the derived entities inhere union attributes, which are the address and the pure virtual operation "`sizeof`". The following union `u2` is the subtype of `u1` because all of its members are also members of `u1`.

```
union u2 {int a; date c;}
```

Union's member can be unnamed. Below is the alternative declaration of `un`.

```
union un {int;float; date;}
```

6.12.4. Enum

Enums are restrictions of the base type to a list of admissible values. Consider the C++ enum

```
enum currency {dollar, frank, euro};
```

In this language, it will be interpreted as a restriction of the type `int`, which is named `currency` whose only admissible values are values `0,1,2` named respectively as `dollar`, `franc` and `euro`. In ***T***, this enum has to be represented as

```
enum currency (int){ dollar, franc, euro};
```

Difference between ***T*** enum and C enum consists of the treatments of values. Thus, the value of type `currency` can be both the named value as `dollar` and the unnamed value as 0 whereas C enum allows only constant names. Following code is correct ***T***, but wrong in C/C++:

```
my_currency=0;          //true for T, wrong for C++
```

A definition written in the way of the above C++ enum without explicit base

```
enum currencyName {dollar, franc, euro};
```

enumerates three lexical words, which could be associated with a particular meaning somewhere after the definition point.

A restriction can also be defined implicitly without the keyword "*enum*". The following three declarations of `my_currency` are functionally equivalent.

```
currency my_currency;
int my_currency =| currency;
int my_currency =| {dollar=0, frank=1, euro=2};
```

The list of values can be alternatively given as:

```
int my_mycurrency =| {{dollar, frank, euro} = {0..2}};
```

Enum’s basis can include several components, which can be reused separately. Thus, the union below represents correlations between a month number and its name:

```
enum month (int number, string name){1 “Jan”; 2 “Feb”; ..; 12 “Dec”;}
```

Correlations allow specifications of the mutual relations between different parameters of the same procedure. Consider the function `write_month`, which gets the number of a month, returning the string with the month name.

```
proc write_month(int number, out string name);
```

The following version of this function contains the specification of the dependencies existing between input and output parameters.

```
proc write_month(month.number number, out month.name name);
write_month(1,”Jan”);            //OK
write_month(1,”Feb”);            //Wrong, because input parameter 1 requires January
write_month(3);                  //Short form of write_month(3,”March”);
write_month(,”March”);           //Short form, input parameter omitted
```

A reference to an object can be restricted as well. Following union confines the admissible values of a reference to an `int` object

```
int ai,bi,ci;
enum UOnlyAB int&{ai,bi};       //UOnlyAB is a reference to int which can refer only to ai or bi.
UOnlyAB onlyAB;
onlyAB:=ai;                      //OK
onlyAB:=ci;                      //Wrong, only ai and bi are allowed to be referred to by onlyAB
```

6.13. Procedures, Functions, Events, Rules

A procedure is a type of event. In general, an event has a structure (body), which may be omitted if unnecessary. A structure of an event can be defined through another event (in case when both events are synonyms) or as a collection (sequence or parallel set) of enclosed events. Following are examples of procedure’s definitions.

```
proc simultan()            //The procedure simultan consists of the simultaneous occurrence of one and two
{{
    one();
    two();
}}
proc sequent()             //The procedure sequent consists of consequent occurrence of one and two
{
    one();
    two();
}
proc sameAs() sequent(); //The procedure sameAs  is the synonym of sequent
```

Unnamed procedures are also allowed.

```
proc <<choose>>(Article article)...;    /the string in the double angle brackets is a comment
```

Functions are procedures producing some object. While being semantically completely similar to procedures, functions allow distinct grammar for representation of their occurrences. A function `plus` returns the sum of two integers

```
func plus int(int,int);
proc plus(int,int,out int);      //this is the procedure equivalent of the above definition
plus(3,5,8);                     //designation in procedure style
```

```
plus._r(8)(3,5);              //designation by intermediary of its output parameter
plus(3,5);                    //this designation is equivalent to  plus._r(8)
#plus(3,5)                    //a command requiring execution of plus
```

An event is an occurrence of a procedure or function. It consists of a change or a sequence of changes and inclusive events.

```
proc  prc();                  //declaration of the procedure prc
prc();                        //unnamed event of prc
prc() prc2;                   //the occurrence of prc named prc2
event myStory{…};             //the specification of an event. Mystory
myStory;                      //the narrative reference to an event.
```

If the procedure `prc` is defined in an application `usp` it has to be referred outside of this application as “`usp::prc(p1,p2..,pn)`” or as “`usp  prc(p1,p2..,pn)`” if it stays at the sentence’s begin.

A rule is an executable equivalent of a procedure (function).

```
proc prc();
rule com{prc();};
#prc();                       //command requiring execution of prc
com();                        //the same as before
^com();                       //indicative text speaking about occurence of com is(was) executed somewhere
rule com1 int(int);           //declaration of a executable function.
rule com2 (int,out int);      //the same as before
```

In general, procedures, functions and events are involved in the following language contexts

- A declaration of a procedure, function, event.
- A definition of a procedure, function, event.
- A reference to a procedure or its components.
- An event (occurrence) of a procedure, function.
- Occurrence of an event (defined by its definition)
- A reference to an event or its components.
- Commanding activities

The formal parameters of a procedure are specified in the comma separated parameter list. Formal parameters are essentially public objects known inside and outside of the procedure’s body. A formal parameter can be either a value parameter or a reference parameter.

A value parameter is a state of an object used in the procedure’s body. Thus, parameters of C/C++ functions are regarded as the values of function’s variables created immediately after the function’s start. Consider following C++ example

```
//C++ code
//first file
void Modulo11(int& res, int number)
{
     int rest = number / 11;
     res = number - rest;
}
```

```
//second file
void Modulo11(int&, int);
extern int reslt;
extern int num;
Modulo11(reslt, num);      //number gets the value of num
Modulo11(reslt, 256);      //number gets the value 256
```

In C++, the variable `res` is the pointer to an `int`, in this case a pointer to `reslt`, the parameter `number` is an automatic variable created at the start of `Modulo11` and initialized by the externally set value. The same executable code can also be defined in ***T***.

```
cfunc Modulo11(int &res, int number);
```

A non-executable procedure `Modulo11` can be declared in a number of ways

```
proc Modulo11 ( int &res, local int ); //a non-executable version of above declaration
extern int num;
proc Modulo11 ( int &res, num );       //only values of the object num can be used
proc Modulo11 ( int &res, $int$num ); //just the same; only in the shorter form
```

The second parameter can also be omitted

```
Modulo11(reslt);                      // the value of variable num is unclear and is ignored
Modulo11(reslt, num);                 //the same as before
Modulo11(reslt, 256);                 // the occurrence with num equal to 256
Modulo11(reslt, $int$num2);           //Wrong! Only the values of num are allowed
```

The main purpose of value parameters in ***T*** consists in eliminating of side effect occurring when some procedure uses values of objects defined outside its body.

In general, the indicative occurrence of some function can be designated in a number of ways.

```
proc prc1(int p1,int p2);
prc1(3,4);                  //conventional positional list
prc1(.p2(4),.p1(3));        //dot with following word references parameter through its name
```

In case of mixing positions with names in the same list, positional parameters have preference.

```
prc1(.p2(5),4);             //means prc1.p2 = 5, prc1.p2 = 4.  prc1.p1 gets  no value
```

In the indicative code, the actual parameters may be expressed partially.

```
proc prc3 (int n1,int n2,int n3);
prc3(3);                    //OK
prc3(3,,5);                 //OK
#prc3(3,4,5);               //OK, an imperative invocation of prc3
#prc3 (3,,5);               //wrong, the imperative code needs setting of all actual parameters
```

6.14. Information Statements

A literal structure called **information statement** (abbr. info statement) is basis construct used for representation of adjectives and adverbs.

An info statement represents the complete information associated with the execution of a comparison routine. Its syntax is similar to a function call extended by the result of a comparison as its first parameter. Thus, the function `isEqual` allows two info statements

```
int  num1, num2;
```

```
proc isEqual(out bool, int, int);

isEqual(true,  num1, num2);     // this info sentence means  "num1 is equal to num2"
isEqual(false, num1, num2);     // this info sentence means "num1 is not equal to num2"
```

If the comparison routine produces a unique value, the procedure name can be replaced by the name of this value.

```
enum equility int{equal, nequal};
proc isEqual(out equility e, int, int);
nequal(num1, num2);                //this indicative sentence means that   "num1 is not equal to num2"
```

Yet another possibility is to define lexical enum listing a sequence of ***T*** words.

```
enum result {<, ==, > };         //signs "<", "==", ">"  are admissible names in T
func compare result(int &first, int &second);
```

Depending on the input values, `compare` returns "`>`", "`==`", "`<`". Accordingly, its occurrence can be designated with the help of the following construct

```
>(a,b);                          //means compare( a,b) produces ">"
<(a,b);                          //means compare( a,b) produces "<"
==(a,b);                         //means compare( a,b) produces "=="
```

The next step of simplification of the above notation is possible with a special form of a comparison routine returning a lexical word. Its syntax is

```
info <returned enum>(<param1> [, param2,...paramn])
```

This kind of info statement can be written in the function-like notation for every number of parameters, or as a binary operator in case of two parameters and as a unary operator for a single parameter. The function `isEqual()` can be redefined as follows:

```
info  {equal,nequal}(int,int);//equal, unequal are legal lexical words
num1  equal num2;                //the indicative sentence means "num1 is equal to num2"      binary operator
num1  nequal num2;               //the indicative sentence means  "num1 is not equal to num2" binary operator
equal (num1, num2);              //the indicative sentence means "num1 is equal to num2"      functional notation
nequal(num1, num2);             //the indicative sentence means  "num1 is not equal to num2" functional notation
```

The statement with one input parameter can be written in a function-like notation or a unary operator in a postfix or prefix form. The default form is prefix. The following info routine has to be produced during the comparison between the actual grass color and the predefined set of available colors.

```
enum Color (black, white, green, brown, yellow, red, orange};
info Color(Object);
into Color(out, Object);        //the same as before
green grass;
green(grass);
```

An unary postfix operator needs to be declared as:

```
info Color(Object, out);        //second variant
grass green;
```

6.15. Associations and Attributes

The common way of representing entities like a person consists of defining the class, whose components are a person's attributes like age, position, clothing size etc. While this is very effective way of defining distinct entities, it is incorrect from the viewpoint of actually existing relations between persons and their attributes. The only parts of a real person are the person's body parts and internal organs while all aforementioned attributes are external objects related to this person in one way or another.

T provides the concept of association that allows referring to entities by their features and affiliations as "his mood", "the smoking pipe of Hemmingway", "a house in London". Associations are the most fundamental kind of relationships. All other relationships like relations between an object and its parts, between a class and its method are based on this concept.

Associations are defined as relations between entities defined on the same level. The objects `name` and `shirtSize` are associated entities:

```
string name;
char shirtSize;
```

The relations between these entities can be expressed using the following two constructs:

- The horizontal qualifications between associated elements refer to the object `shirtSize` as `name@shirtSize`.
- The association constraint is used for expressing dependencies existing between association members. Thus, the meaning of the phrase "Peter's shirt size is 'M'" can be expressed as `name("Peter")@[shirtSize(M")]`;

Alternatively, associations can be defined in the form of attributes, which reference the associated object only through the head entity. The following is the association between `name` and `shirtSize` expressed in the form of attributes.

```
string name [char shirtSize;];
name;                              //OK
name@shirtSize;                    //OK
shirtSize@name;                    //Wrong. shirtSize is not defined as a sovereign entity.
```

The number of attributes is unlimited and many of them can be referred to in the same sentence.

```
string name [char shirtSize; Date birthDay];
name(Robert)@[shirtSize("M");birthDay(1.1.1980);]
```

In most cases, associations are used for expressing dependencies between objects participating in the same algorithm. Consider the following example:

```
class Person {public: string name; char clothSize;...}
class Cloth  {public: string clothKind; char clothSize;...}
proc measureClothSize(Person &person, Cloth &cloth, out char size);

proc buyCloth(Person &buyer, Cloth &cloth);
...
```

```
extern Person aPerson;
extern Cloth  aCloth;
```

There are several ways to express the information "Peter bought a shirt or size M". One is to define a respective info statement.

```
measureClothSize(Peter, Shirt, 'M');
```

Another way consists of redefining `measureClothSize` as an unnamed procedure and use its paramters in the association statement.

```
proc (Person &person, Cloth &cloth, out char size); //the unnamed procedure is defined as
                                                    //a synonym of measureClothSize

Person(Peter)@[cloth(Shirt),size("M")];             // first specification variant
Person(Peter)@cloth(Shirt)@size("M");               // second specification variant
```

Below specifications are made with the help of object's constraint (see section 6.18 for details).

```
aPerson.{name="Peter";clothSize="M";} // means: "aPerson's name  is Peter, its closeSize is M"
```

Alternatively, the same meaning can be expressed as

```
aPerson.{name="Peter"}.clothSize=='M';
```

Following phrase expresses the meaning "Peter bought a shirt of the size M".

```
aPerson.{name="Peter"}@[aCloth.{clothKind("shirt"); clothSize("M") } ]
```

"A sheet of the size M was bought by Peter". This phrase has almost the same meaning as before but still defers in details

```
aPerson.{name="Peter"}@aCloth.{clothKind("shirt"); clothSize("M")};
```

Another version of the above sentence.

```
aPerson.{name="Peter"}@aCloth. clothSize("M")@clothKind("shirt");
```

This method of expressing the same associations consists of explicitly binding them to the respective algorithms.

```
measureClothSize.person.{name="Peter";clothSize="M"};
buyCloth.{ buyer.name("Peter");cloth.{clothKind("shirt");clothSize("M") } };
```

The last sentence however looks pretty unnatural. We don't say "bought shirt of size M by Peter" but rather "Peter bought a shirt or size M". ***T*** allows exactly such expressions referencing a procedure by the name of its actual parameter with the help of "`@@`"

```
aPerson.{name("Peter")}@@buyCloth.{cloth.{clothKind("shirt");clothSize("M") } }
```

6.16. Applications

An application is a code piece qualified by some entity, components of which are visible in this code piece. The entities defined inside the block can be referred with a application object followed by the double colon "::", which can be omitted in certain cases. The qualifying object can be either class or instance.

```
class Obj {int ii; int c};
Obj obj;
```

```
obj::{ii = 4;};
```

All components and methods of an instance are original members of its application. Furthermore, it can be extended by additional references to its original members and procedures whose body referencing original members and methods, starting from or references

```
Obj::
{
    proc someproc(){...};
    int &aref := ii;                //aref is reference to ii from instance of Obj
}
Obj::someproc();                  //OK
Obj::aref;
obj::aref = 3;                    //OK
obj  aref = 3;                    //wrong, a component can only be referred with "::"
obj::someproc();                  //OK
obj  someproc();                  //OK, the same as before
```

A double-colon sign "::" is used exclusively for reference to application components. Differently from C++ it is not used for reference to the class members.

Among other, application are used for representation of *changes*. Changes are elementary procedures representing sequences of object's states. Thus, a clench of a fist is a change of the fist, a turn of a fan is a change of a fan, the working of a motor is a change of a motor, a C++ program is a change of a C++ innate processor. The following code represents grows of an apple during some time.

```
class Apple
{
    int size;
    ...
}
Apple apple;
apple.size = 20;                            //Admit the size in given in mm.
apple.size = 25;                            //The distance between events of
apple.size = 30;                            //size measurements is undefined
```

In ***T***, this change can be represented by the four alternative ways

```
apple::{size(20);size(25);size(30);}        //an occurrence of change
apple::{proc grow(){size(20);size(25);size(30);}//the declaration
proc apple::grow(){size(20);size(25);size(30);  //the alternative declaration
proc apple::grow()size::{(20);(25);(30);}   //yet more optimized alternative declaration

apple::grow();                              //an occurrence of this change
```

In case if the application's name stays at the sentence begin, it can for referencing an event (procedure occurrence).

```
apple grow();                               //OK
apple size=30;                               //Wrong! size is not an event
```

Changes are analogous to verbs in natural languages and can be used in active and passive forms. Thus, the sentence "Bob puts the cup on the table" can be expressed in ***T*** as follows

```
Bob put(table, cup);
```

The passive form of this sentence "the cup is put on the table by Bob" has following syntax:

```
cup + Bob::put(table,@);                //"@" is the predefined reference to the object preceding +
```

Class methods referring no entities outside of this class are its inborn changes.

6.17. Components

A component is a part of some entity.

```
class Obj
{
    int m1,m2,m3;
};
Obj obj;
obj.int comp;                   //comp is some int component of the instance obj
comp := obj.m2;                 // comp designates obj.m2;
Obj obj2;
comp := obj2.m2;                //Wrong comp is not a component of obj2
Obj.int classcomp;              // classcomp can be the part of any Obj instance
classcomp := obj.m2;            //OK, the class component can refer whatever class instances
classcomp := obj2.m2;           //OK
```

Members of a class are inborn components of this class.

6.18. Restrictions

A restriction is an operation of constraining some object.

```
class numbers {int num1,num1;};
class restr:= numbers.{num1 = 1;num2 = 3} // restr is numbers with num1== 1; num3 == 3
restr vl;
numbers nInst1, nInst2;
nInst1.{num1 = 3;};
nInst2 = nInsts1.{num2 = 4;}            // num1 of nInst2 is equal 3, num2 is equal 4
```

Another kind of restriction is the specialization of procedures:

```
proc base(int k);
proc base_val:=base(3);
base_val();   //this is equivalent of base(3)
```

A value is a state of some object. It differs from its owner object by the constrained members or/and new methods, which refer exclusively the methods of the owner. Values can also be introduced as abstract entities without explicit specification of their features. The exact characteristics of the value `null` of the class `bit` are unknown at the declaration point.

```
class  null := bit.{?};
```

6.19. Sentence Format

T uses a number of sentence types including C++ like structures ("*if*", "*while*", "*for*", "*do*", "*switch*", "*case*", "*break*", "*return*", "*continue*", "*goto*"). The main indicative sentence of ***T*** is the indicative equivalent of the call statement of C^{++} with the following syntax

```
action,association,association..;
```

where `association` is between this action and its environment.

The sentence "I started to work during a rainfall" can be written by three ways:

```
I::work()@start@[was;during(rainfall)];
I::work()@start, was,during(rainfall);
I  work()@start, was,during(rainfall);
```

6.20. Specification of a communication environment

The failure of BNF in expressing the meaning of a program results from its restriction to the syntax specification. For instance, if there is a function call statement like "`foo(a,b,3);`" its structure can be described in BNF, but this specification will say nothing the way of processing this statement.

T goes completely different way explicitly referencing the communicators exchanging texts in this language. The set of lexical generators and readers can be referenced through the predefined references `_reader` and `_writer`. The referenced entities can do miscellaneous things, though their only mandatory actions are that concerned with the generation and (or) reading a text composed in this language. The externally accessible function level of a communicator consists of lexical procedure, whose parameters are valid code items. Every code item has its lexical procedure what makes it possible to represent the language items as well as operations with them. See more in (Sunik 2012).

T uses several meta-constructs for representation of the lexical structures. In essence, it is the conventional specifiers denoting functions, structs, unions, enums, arrays with the only difference that its parameters are not simple strings but the lexical items.

The fundamental metaclass `sentence` defines classes of sequentially organized linguistic entities. While its syntax is similar to that of *struct*, member and type names may be omitted. Elements of `sentence` may be separated by one or more blanks, comments and the characters tabulation, new line, and carriage return.

The fundamental metaclass `lexunion` defines a union of lexical element.

The fundamental metaclass `lexer` is used for representation of lexical procedures, which are procedures, whose input parameters consists of the code items.

Let us consider the declaration of a C++ lexical procedure implementing a function call. Differently from C++, ***T*** allows reverse declaration order.

```
lexer implicit call(callSentence);   //The specifier implicit is explained below

sentence callSentence
{
    lexword   functionName;
    sentence
    {
        "(";
        actParList;
        “)”;
    }         parameterList;
}

sentence actParList
{
    parameter par;
    commaList..;
    “)”;
}
lexunion commaList
{
   blanks;
    “,”;
   blanks;
   parameter;
}

lexunion parameter
{
    expression expr;                //is not defined in this text
    lexword name;
    literal lit;                    //is not defined in this text
};

typedef “ ” blank;

union blanks
{
    void;
    blank..;
}      //optional blanks

sentence lexword                   //letword is a sequence of letters with a length of 1 to 20 bytes
{
    letter;
    letunion
    {
        letter;
        digit;
        “_”;
    }.. rest[0..19];               //rest is a string, which can contain from 0 till 19 letters or digits.
}
```

The specifier `implicit` causes omissions of the `lexer`’s name from the call statement. So the call as “f`oo (3,a)`” is correct. Without this specifier, this lexer has to be called as “call `foo (3,a)`”

The class `letter` represents ***T*** letters. It is the subclass of `lexchar` representing characters of the ***T*** alphabet.

7. Example representations

This chapter is completely devoted to examples of non-executable representations, which are differed by their goals and composition rules from executable representations (programs). Essential differences between these two kinds of code can be seen in the following comparison.

1. Purpose*.*

Executable code. Programs are sequences of instructions controlling behavior of code readers, which actually executes the actions designated by the code. The command "put the book on the table" means among other that a receiver of this message is able to take the requested book and put it in on the mentioned table.

Non-executable code. Narratives are pure data informing their readers about something. Generally, they do not require executable activity from the side of code readers. The sentence "the book is put on the table" implies no spatial, time or logical relations between its reader and the depictured process.

2. Completeness.

Executable code. A program is a hierarchical structure of subroutines (functions), the lowest level of which consists of the low-level machine code executed by the physical computer's processor. In order to execute a program, a computer has to know how to execute each its subroutine, i.e. it has to possess all implementations of all subroutines referred throughout execution. Even a single unrecognized machine code instruction will stop program's execution and the same will occur after single invocation of a missing subroutine.

Non-executable code. In general, narratives require only prototypes of actions, while their implementation details are needed in rather exceptional cases. The phrase "the flight c367 is landing" delivers sufficient information to the persons awaiting this message, while detailed description of landing is necessary only in extraordinary cases.

3. Directness is a consequence of completeness.

Executable code. The only relevant characteristic of an executable procedure `foo` is its implementation. The same is true for the software objects..

Non-executable code. The characteristics of environment are oft more important in non-executable code than those of a referred entity. Thus, the phrase "Bob is born" designates an event of the procedure `give_birth` but the detailed algorithm of this procedure is hardly of any interest in general case, while indirect associations like the date and the place of birth has to be required much more frequently.

Another example. The sentence "The travel was long and tiring" said nothing about the travel itself, but only about feeling associated with it and the time span it took.

4. Type-centration.

Executable code. Implementation of instances tend to be defined in their types (classes and procedures), which normally express the complete understanding of the represented algorithm. Despite distinct instances of classes could be processed differently this is often viewed as the bad programming style.

Non-executable code. In general, each event can possess its own implementation. A procedure in non-executable code is normally an abstract procedure characterized by its parameters and attributes, while the detailed specification of its algorithm can be completely different for each its occurrence or a group thereof. Thus, events "Bob is born" and "John is born" refer to the same procedure `give_birth` but these events can be such different that they will require distinct descriptions of each case.

5. Sequentiality.

Executable code. Programs are basically consecutive because their statements are processed by the same processing unit by default. Parallel algorithms are represented as sets of consecutively executed programs.

Non-executable code. Most algorithms occur in parallel while sequential algorithms are rather exception. "Roses grow in the garden" means the parallel grows of many roses on a particular place.

The presumption of this work is that the ontologies of event and time defined on the base of other approaches (Clark and Porter 1999; Oaklander 2004; Smith, Welty, and McGuiness 2004; etc.) are expressible in terms of the commonsense ontologies below.

7.1. Parts of Speech

Adjectives and Adverbs are represented by info routines.

```
info (fair, unfair}(Object);
person Bill;

fair Bill;
```

Prepositions about, *below*, *from*, *out* are implemented with info statements. The following info statement characterizes the relationship between an entity and a container

```
info {from, none} (Object part, Object obj);
struct S{int i1,i2;  float f;}      //the container S includes i1, i2 and f
float from S;                       //designates "f from S"
int.. from S;                       //designates "i1 and i2 objects from S"
```

Quantifiers as *all*, *some*, *few* are represented with info routines. They are essentially relations between a heap and its subset.

```
info {all, many, some, few, undef} <typename T>(heap<T> subset, heap<T> plurality);
```

```
Student.. campusStudents;
Student.. &distinguishedStudens;
Student.. &commonStudens;

distinguishedStudents few campusStudents;
commonSstudents many campusStudents;
```

Using the preposition *from* allows following definitions.

```
Object my_university;                   //includes buildings and persons
// unnamed heap including all students from my_university
Student.. from my_university;
 // unnamed heap including some students from my_university
Student.. some Student.. from my_university;
// heap with a name stdnts, includes some students from my_university
Student.. some $Student..$allStdnts from my_university;
// the same as before
Students.. some allStdns;
```

Verbs designate events (occurrences) of procedures. The basic construct used for expression of procedures is an application.

```
proc apple::grow(){size(20);size(25);          //the declaration
apple::grow();                                 //an occurrence of this change
```

Verbs can be used in active and passive forms Thus, the sentence "Peter puts the cup on the table" can be expressed in ***T*** as follows:

```
Pete::put(table, cup);
```

The passive form of this sentence "the cup is put on the table by Pete" has the following syntax:

```
cup, Pete::put(table,_noun);                   //_noun is the predefined reference
```

The syntax of a sentence can be found in 6.19.

Pronouns are represented as references (temporary names) used to refer to an object. Such nouns as *neighbor*, *patient*, *chief* etc. are also references to real objects.

Pluralities are intended to represent the plural form of human languages. The phrase "children plant trees" will be translated to ***T*** as follows

```
child..::plant.. (tree..);
```

7.2. Events

An event is an occurrence of a procedure. It consists of a change or a sequence of changes and inclusive events.

```
proc prc();                          //declaration of the procedure prc
prc();                               //unnamed event of prc
```

A change consisting of similar states represents a process of perpetuation of something, e.g., "a cat on a mat" means the place of the cat is not changed during observation time.

Representation of events essentially depends on the code form. While the event implementation (algorithm) is important in the imperative code, it is normally not true for the narrative representation. The general way of narratives consists of expressing relations between specified and related events. For example, the phrase "he came home at the end of the day" denotes a coincidental time between the process of someone coming home and the day ending occurring at this moment. The event (the day ending) is considered as being the longer of both processes. So the exact semantics of this phrase is that an event of coming home occurred somewhere *during* the ending of the day.

The phrase "I was at home" implies a connection in time of the event of me being at home relative to the event of producing this phrase.

The phrase "the train leaves at 15.00" means that the train departure occurs simultaneously with the clocks showing 15.00.

The phrase "Bill drove 15 hours" in turn means that the duration of Bill's drive lasted the same amount of time as a clock needs to count 15 hours.

The declaration below introduces essential attributes of an event, which relate to the designated event in this or another way. The type `event` is a basic type, its definition is impossible without a recursion, so it is essentially a pseudo-declaration. The predefined reference `it` is used in the brackets for referring to the master object. The reference `it.it` allows referencing the master object of the encircling level and so on. The master object is the default first parameter.

```
proc event
[
    //Associated events start and finish are parts of the main event.
    event &start;                    //start of the main event
    event &finish;                   //finish of the main event
    event &occur;                    //synonym of the main event
    //An event ev is an independent event occurring simultaneously with the main event
    info {during, before, after}{event ev2);
    info (was,is,will}{();
    info (at,false}{event ev1);
    info {lasts,false}(event ev1);};
    info {before,during,after}(event ev);
    //event enclosing includes the main event
    info {in,false]{event enclosing)
    [
        //the following code define relations between respective events
        ( enclosing@start before  it.it@start  ||
          enclosing@start  during it.it@start)
        &&
        ( enclosing@finish after  it.it@finish ||
          enclosing@finish during it.it@finish );
    ]
    //following procedure compares the main event this with the time of the text production
    Place place [it.it on it;];        //the place of events occurrence
]
```

Grammar tenses establish a relationship between the time of producing the sentence by a code writer and a time of an event. The info statement registering the event's tense can be defined as:

```
info (was,is,will}{event ev)
{
    extern event _write;
    if( _write during  ev )
        return be;
    elsif( _write after ev )
        return was;
    else return will;
}
Bob::goto(cinema),_v was;                 //means “Bob went to a cinema”
Bob::walk@[be]();                         //means “Bob walks”
```

Grammar tenses can be redefined as references in the body of the class `event`.

```
class event: proc
{
    literal _w  := was(this);             //past time
    literal _wi := will(this);            //future time
    literal _p  := is(this;               //present time
}
Foo()._w;                                 //this event of Foo occurred in the past
```

7.3. Time

A time (date-time) is a state of a clock. A clock is a device implementing four general procedures: `count`, `stay`, `reset`, `stop`. The procedure `count` refers to the counting procedures `Second`, `Minute`, `Hour`, `Day`, `Month`, `Year`, `Era`. The value stored in `seconds`, `minutes`, `hours`, `days`, `months`, `years` represents the current clock‘s time.

A state of a clock is an event of persisting a current state. The event finishes with the change of the smallest component (e.g., the state 1.1.2008 13:30:31 ends when a number of seconds changes to 32).

```
class Time
{
public:
    //type integer is the language number consisting of digits
    integer years,
            months,
            days,
            hours,
            minutes,
            seconds;
    time(){years = months = days = hours = minutes = seconds = 0;}
    time(integer seconds = 0, integer minutes = 0,integer hours= 0,integer days= 0,
         integer months = 0,integer years = 0);
}
class Clock;
//the following procedures are applications of a Clock
Clock::{
public:
    Time<<>>;                                  //elements of unnamed object can be referenced directly

    proc Second ();
    proc Minute (){for(seconds = 0; seconds < 60; seconds++)      Second();}
    proc Hour   (){for(minutes = 0; minutes < 60; minutes++)      Minute());}
    proc Day    (){for(hours   = 0; hours   < 24;   hours++)      Hour();}
    proc Month(integer ndays){for( days=0; days < ndays; days++) Day();};
    proc Year(bool leap_year = false);
    proc Era()    {for(years = 0;;years++)  Year(years / 4 * 4 == years)}
    proc count()  {Era();}              //the counting procedure
    proc stay();                        //the staying (non-counting) procedure
    proc reset()                        //resets time to null
    proc stop();                        //the stopping procedure
}

proc Clock::Year(bool leap_year = false)
{
    months = 1;     Month January(31);
    months = 2;     Month February(leap_year ? 29 : 28);
    months = 3;     Month March(31);
    months = 4;     Month April(30);
    months = 5;     Month May(31);
    months = 6;     Month June(30);
    months = 7;     Month July(31);
    months = 8;     Month August(31);
    months = 9;     Month September(30);
    months = 10;    Month October(31);
    months = 11;    Month November(30);
    months = 12;    Month December(31);
};
```

The event `Ptime` represents the passing time, which lasts the same time span as the attribute `occurred`.

```
class Ptime
{
publiic:
    Time ptime;
    extern Clock clk;

    Ptime(event ev_begin, event ev_end)
    {
        Time t1 = clk.count()@[it at ev_begin);
        Time t2 = clk.count()@[it at ev_end;];
        ptime = t2 - t1;
    }
    Ptime(event occurred)
    {
        Time t1 = clk.count()@[it at occurred@start);
        Time t2 = clk.count()@[it at occurred@finish;];
        ptime = t2 - t1;
    }
}
```

The class `Ctime` represents the current date-time, which has passed since the start of A.D.

```
event first_day_of_A_D;                 //the first day of A.D.
class Ctime
{
public:
    Time ctime;                         //Ctime is the current date-time shown by some clock
    Ctime(event ev)
    {
        ctime = $Clock$.count()@[it at ev@start] -
                $Clock$.count()@[it at first_day_of_A_D@start];
    }
}
```

The class `age` represents the age of an object at a query-point.

```
class age Time.years
[
    Object   aged;                      //the object whose age is queried
    Ctime    querypoint;                 //the time of the query
    Ptime    pt(aged@creation),querypoint);
    it == pt@years;
]

I walk(),during(Ptime(.hour(3)));       //I am walking during three hours
I walk@[during(Ptime(.hours=3))];       //the same
I walk()@start,at(Ctime(13,15,15,11,2003));//I started to work at 13:15 15/11/2003
I walk(),finish@[at(Ctime(13,15))];      //the same as before
```

7.4. Translation Examples

Below is the translation of several English phrases into ***T***. The phrases used here are originally defined in (Sowa 2010 Thematic Roles). They also can be found at http://www.jfsowa.com/ontology/thematic.htm

Example: "Diamonds were given to Ruby".

```
class diamond;
human Ruby;
proc object::give(entity what, entity whom);
diamond... + some::give._w(@, Rubi);
```

Example: "Mary waited until noon".

```
proc human::wait(event ev)           //wait period
human Mary;
event noon;

ev till noon@start;
ev was;

Mary wait._w(event@till(noon@start) );//alternative
```

Example "Mary waited three hours".

```
Mary wait() ev;
ev@[was, during hours(3)];
```

Example: "The truck was serviced for 5 hours".

```
Worker someone;
class Track;                         //dictionary definition
proc Worker::service(object served);
track + someone::service._w(track(), verb during( time(.hours(5) ) );
```

Example: "Vehicles arrive at a station".

```
class Place;                         //dictionary definition
class Vehicle::
{
    proc arrive(Place target)
}
class Station : Place;
Vehicle.. arrive.. ($Station$);
```

Example: "The key opened the door".

```
class Lock
{
    class Keyhole{...};
    Keyhole keyhole;
    ....
}
```

```
class Door
{
    Lock lock;
    ....
};
class opened_door Door.{...};        //the Door state when it is opened
class closed_door Door.{...};        //the Door state when it is closed
class locked_door Doo.lock.{...};    //the Lock locked the Door
class unlocked_door Doo.lock.{...};//the Lock unlocked the Door

Door::
{
    proc locking();                  //the process of door locking
    proc unlocking ();               //the process of door unlocking
    proc opening();                  //the process of door opening
    proc closing();                  //the process of door closing
};
class  Key::{
    proc insert(Lock.Keyhole);
    proc turn(enum Direction{left,right});
};
Object::
{
    proc put(Object what,Hole into);
    proc push(Object what);
}
Door door;
Key key;

key + some::put(@,door.keyhole);
key turn(right);

if( some::push(door) ) door::opening();
```

7.5. Translation of a big sentence.

The following English sentence is translated to ***T***: "Back in 1969, when Grazyna Bialowas was 18 years old, she and a friend planted two rows of trees to beautify the state-owned cable factory that was the center of her world in the town of Ozarow, outside Warsaw."

The most important thing needed for the translation is a vocabulary defining notions referred to in the translated text. This sentence refers to a lot of concepts such as *place*, *time*, *event, age, beautifullness, center, world, outside, ownership, tree, row, factory*. The complete definition of these notions requires a place far exceeding the size of this article, however the example might well be served also by the "light" definitions exposing only properties referred to in the example.

Each definition may contain two parts: the definition of components and the definition of attributes characterizing its environment. Most of the following definitions contain only the second part, similar to the definitions carried out on the natural language. Thus, the notion of *place* is defined by characterizing its attributes such as `intime` anchoring a place to the particular time and `inplace` representing the enclosing place.

```
class Place {/*..*/}
[
    event intime;
    Place inplace;
]
```

The class `object` is characterized by its `location` and its `owner`.

```
class object {/*..*/}
[
    Place location;
    entity owner;
];
```

The beautification criteria is defined with the help of two procedures. The first one decides whether this place is better/uglier than the other, and the procedure `beautify` defines the beatifying process.

```
info <<beauty_comparison>> {beauteous, ugly} (Place place1,Place place2);
proc beautify(Place bp,event beautification)
{
    bp@intime(event@[after,beautification]),         //bp after beautification
         beauteous,                                  //is beautifully then
         bp@intime@(event@[before,beautification]);//before it
}
```

The following classes declare other entities referred to in the subject sentence

```
class human : object {/*...*/};
proc human::implant(plant);
human Grazyna, a_friend;

Place cable_factory [=:object@@owner(State)];
class World : Sphere {/*..*/};
class Town  : Place  {/*..*/};
Town Ozarow;
```

The reference `then` is defined as the reference to the event occurring in parallel with another event. For the sake of space, I omitted the complete definition of `then`, which is the past event.

```
class then :event [event occurred];
```

The following info statement represents the criteria of being outside

```
info {far,near} (Place mainplace,Place otherplace);
```

The event `Grazynas_act` is the action performed by Grazyna and a friend.

```
event Grazynas_act  [during( Grazyna@age == 18)]
{
    {Grazyna,a_friend}.implant.. (tree.. @[ from row<tree>..(2) ]);
        //the unnamed event "age == 18 == true" is given directly not as a reference
}
```

Below is the translation of the sentence components

```
then(event@[at(.years:=1969)]),
Grazynas_act@[it=:beautify.beautication;was,place(cable_factory)];
```

Cable factory is the center of her world.

```
World@[owner(Grazyna)].center(cable_factory);
cable_factory@inplace(Ozarow=:otherplace@[mainplace(Warsaw),far]);
```

The whole sentence can be represented by a single ***T*** sentence as follows.

```
then(event@[at(.years:=1969)]),
 { {Grazyna,a_friend}::implant.. (tree.. from row<tree>..(2) }
   @[
        during(Grazyna@age == 18);
        it=:beautify.beautification;                    //it is the implicit reference to
        was;                                            //the master event
        place
        (
            cable_factory @
            [
                World @[owner(Grazyna)].center(it),    //cable_factory is a center
                inplace (Ozarow@[near Warsaw])
            ]
        )
    ];
```

8. Conclusions

1. The keystone of the Theory of Meaningful Information (TMI) is the definition of information, which can be applied to information of every kind, level and complexity. This definition actually introduces the uniform view of all entities, which are able to produce and use information. This view creates the basis for the formal representation of all information (knowledge) related tasks.

2. Knowledge in TMI is viewed as the internal information content of an Information Capable Entity (ICE) that reflects features of an ICE's environment, providing abilities for flexible reaction of the environment's alteration. Any living creature or artificial device that is able to acquire and use knowledge can be viewed in this way. The most powerful of all known ICE is a human being, whose information related capabilities essentially supersede those of all other ICE. ICE modeling human beings is designated in this work as Homo Informaticus (HI).

3. Model of HI perceives a human being as a self-programming system commanded by the controlling computer (brain). The body of an HI is seen as the complex set of processing units, each consisting of a muscle attached to a bone or an internal organ. A muscle executes one of two commands contract or relax, which, in turn, causes a movement of a bone or organ. Another important part of this system are the senses, which consist of multiple receptors interpreted as volatile variables functioning similar to the input ports of conventional computers.

Human knowledge is represented by the programming code and data (perceptions) of an HI. The model provides the method of representation of overall human knowledge starting from low-level perceptions that humans acquire during their lives and continuing until encompassing the high-level knowledge. Various kinds of knowledge are interpreted as programs of distinct levels. The built-in algorithms of the lowest level are unconditional reflexes and basic skills, the low-level soft programs designate skills, and the algorithms of the high level stand for beliefs, concepts, etc.

The intellectual operations performed by the HI's brain are interpreted as manipulations with the code of these programs, and a natural language is considered as the system enabling programs' exchange between various individuals. Only programs of a high level can be exchanged in a natural language, other programs constitute tacit knowledge, which if at all can only be transmitted with the help of immediate learning when one get new skills by copying behavior of another HI.

The development of a human being from a baby to a mature individual is viewed as the process obeying the general law of software development: "*the more software is implemented* *on a certain system, the more powerful this system is*". At the beginning, there is only built-in functionality, which is used for producing elementary programs, which, in their turn, constitute building stones of the high-level entities.

4. The model of HI pretends to be adequate because it represents the real features and behavior of a human being.

- A human is born with only inborn reflexes, but without knowledge, which is acquired throughout its life. The presence of the inborn knowledge is naturally explained by this model. (All programming devices have built-in functionality, which is used as the basic for creating complex programs.)
- The complete knowledge structure is explained by this model.
- presence of skills, tacit knowledge, high level knowledge,
- changes as the basic component of the knowledge
- The secondariness of the language knowledge what shows in the exchange of only previously known information
- Practical ability to present whatever semantics
- Formal representation of thinking process

5. The part of TMI devoted to the Theory of the Universal Representation Language defines the way of building such a language and actually allows to build it. The universal representation language ***T*** designed on the basis of the aforementioned theory can be utilized for representation of the complete information presented by this model. ***T*** unites the universality of a natural language with the exactness of a programming language. It is employed in the manner of a natural language with the purpose of information exchange between various communicators. The semantics transmitted by its code consists of the conventional knowledge about objects, actions, properties, states, etc.

T is based on the extension of the C++ conceptual system and allows the representation of both executable (imperative) and non-executable (indicative) information and its default code form is indicative. Similar to a natural language, it is not confined to any particular representation domain, implementation, communicator or discourse type.

6. ***T*** can be employed for solving various tasks:

- Representation of the human information of the perception level
- Representation of the high-level information transmitted by natural languages. It can be used for building systems that understand human language as well as the pivot language in systems of automated translation of natural languages.
- Robototechnics, where it can describe the overall functionality of a robot
- Programming, documenting of software and hardware, specification of various standards, and setting of various tasks in programming. ***T*** allows the creation of a

monolingual communication environment in which this language can be used for programming on different levels as well as for generating various non-executable specifications. The full information relating to a particular OS can be exposed in T.

- Embedded systems, where T enables composition of non-executable specifications and programming.
- Expression of whatever knowledge needs explicit expression. Provided there is a corresponding vocabulary, ***T*** can be used to compose any text representing entities from the domains of chemistry, physic, biology, etc.
- The database for knowledge representation. Such a database consisting of non-executable code can be extended unlimitedly and browsed for various purposes.

7. Explicit representation of the information transmitted by natural languages. ***T*** is useful in building systems that understand human language as well as the pivot language in systems for the automated translation of natural languages.

9. Bibliography

1. Alexandrescu, Andrei. 2010. *The D Programming Language*. 1 edition. Upper Saddle River, NJ: Addison-Wesley Professional.
2. "Artificial Intelligence." 2018. *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Artificial_intelligence&oldid=829204842.
3. Backus, J. W. 1954. "The IBM 701 Speedcoding System." *J. ACM* 1 (1): 4–6. https://doi.org/10.1145/320764.320766.
4. Backus, J.W. 1959. "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference." In *ICIP Proceedings*. Paris. http://www.softwarepreservation.org/projects/ALGOL/paper/Backus-ICIP-1959.pdf.
5. ———. 1977. "Can Programming Be Liberated From the von Neumann Style?" In . http://www.thocp.net/biographies/papers/backus_turingaward_lecture.pdf.
6. Best, Ben. 1990. "The Anatomical Basis of Mind." 1990. http://www.benbest.com/science/anatmind/anatmind.html.
7. *Britannica Encyclopedia Ultimate Reference Suite*. 2009.
8. Broom, Donald M., Hilana Sena, and Kiera L. Moynihan. 2009. "Pigs Learn What a Mirror Image Represents and Use It to Obtain Information." *Animal Behaviour* 78 (5): 1037–41. https://doi.org/10.1016/j.anbehav.2009.07.027.
9. Burgin, Mark. 1997. *FUNDAMENTAL STRUCTURES OF KNOWLEDGE AND INFORMATION: REACHING AN ABSOLUTE Ukrainian*. Academy of Information Sciences, Kiev.
10. Cellan-Jones, Rory. 2014. "Hawking: AI Could End Human Race." *BBC News*, December 2, 2014. http://www.bbc.com/news/technology-30290540.
11. Chappell, Timothy. 2013. "Plato on Knowledge in the Theaetetus." Edited by Edward N. Zalta. *The Stanford Encyclopedia of Philosophy*. http://plato.stanford.edu/archives/win2013/entries/plato-theaetetus/.
12. Chomsky, Noam. 1953. "Systems of Syntactic Analysis." *The Journal of Symbolic Logic*, September.
13. ———. 1956. "Three Models for the Description of Language." *IRE Transactions on Information Theory*, September.
14. ———. 1957. *Syntactic Structures*. Mouton.
15. Chu, Yaohan. 1978. "Direct Execution In A High-Level Computer Architecture." In , 289–300. ACM Press. https://doi.org/10.1145/800127.804116.
16. Clark, Peter, and Bruce Porter. 1999. "KM - The Knowledge Machine 1.4.0: Reference Manual." 1999. http://www.cs.utexas.edu/users/mfkb/manuals/refman.pdf.
17. Codd, Edgar F. 1972. "Relational Completeness of Data Base Sublanguages." *Database Systems: 65-98*, 65–98.
18. Cole, David. 2014. "The Chinese Room Argument." Edited by Edward N. Zalta. *The Stanford Encyclopedia of Philosophy*. http://plato.stanford.edu/archives/sum2014/entries/chinese-room/.
19. Cooman, Gert de, Da Ruan, and Etienne E Kerre, eds. 1995. *Foundations and Applications of Possibility Theory: Proceedings of FAPT '95 : Ghent, Belgium, 13-15 December 1995*. Singapore; River Edge, NJ: World Scientific.
20. Copeland, Jack. 2000. "A Brief History of Computing." http://www.alanturing.net/turing_archive/pages/Reference%20Articles/BriefHistofComp.html.
21. Crevier, Daniel. 1993. *AI: The Tumultuous Search for Artificial Intelligence, Pp. 44–46.* New York: BasicBooks.
22. Cytron, Ron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph." *ACM Trans. Program. Lang. Syst.* 13 (4): 451–490. https://doi.org/10.1145/115372.115320.
23. Davis, Randall, Howard Shrobe, and Peter Szolovits. 1993. "What Is a Knowledge Representation?" *AI Magazine* 14 (1): 17–33.
24. "Definition of Belief." n.d. *Merriam-Webster*. Accessed March 15, 2015. http://www.merriam-webster.com/dictionary/belief.
25. Devlin, Keith. 1995. *Logic and Information*. Cambridge University Press.
26. Ellis, Margaret A., and Bjarne Stroustrup. 1990. *The Annotated C++ Reference Manual*. Reading: Addison-Wesley Professional.
27. Evans, Roger, and Gerald Gazdar. 1996. "DATR: A Language for Lexical Knowledge Representation." *Computational Linguistics* 22 (2): 167–216.

28. Fleissner, Peter, and Wolfgang Hofkirchner. 1996. “Emergent Information. Towards a Unified Information Theory.” *BioSystems* 38 (2–3): 243–248.
29. Flückiger, Federico. 1997. “Towards a Unified Concept of Information: Presentation of a New Approach.” *World Futures* 49 (3–4): 309–20. https://doi.org/10.1080/02604027.1997.9972637.
30. “Formal Language.” 2003. *SIL Glossary of Linguistic Terms*. https://glossary.sil.org/term/formal-language.
31. Genesereth, Michael. 1998. “Knowledge Interchange Format.” NCITS.T2/98-004. http://logic.stanford.edu/kif/dpans.html.
32. Gorn, Saul. 1959. “On the Logical Design of Formal Mixed Languages.” In , 1–2. ACM Press. https://doi.org/10.1145/612201.612232.
33. Hapgood, Fred. 1982. “Computer Chess Bad - Human Chess Worse.” *New Scientist* 96 (1337): 827–30.
34. Heslin, Cassey. 2014. *How to Start a Hobby in Programming*. MicJames.
35. Hodges, Andrew. 1983. *Alan Turing: The Enigma — Notes by the Author*. Burnett Books/Hutchinson. http://www.turing.org.uk/book/.
36. Hoekstra, Rinke. 2009. “Knowledge Representation.” In *Ontology Representation Design Patterns and Ontologies That Make Sense*. Amsterdam; Washington, DC: IOS Press. http://public.eblib.com/choice/publicfullrecord.aspx?p=471349.
37. “IBM 1401.” n.d. *Wikipedia*. https://en.wikipedia.org/wiki/IBM_1401.
38. “International Civil Aviation Organization.” n.d. Accessed March 4, 2018. https://www.icao.int/Pages/default.aspx.
39. Kennard, Fredrick. n.d. *Thought Experiments: Popular Thought Experiments in Philosophy, Physics, Ethics, Computer Science & Mathematics*. Lulu.com.
40. Kurzweil, Ray. 2014. “Don’t Fear Artificial Intelligence.” *Time*, December 19, 2014. http://time.com/3641921/dont-fear-artificial-intelligence/.
41. Legg, Shane, and Marcus Hutter. 2007. “A Collection of Definitions of Intelligence.” *ArXiv:0706.3639 [Cs]*, June. http://arxiv.org/abs/0706.3639.
42. Levin, Janet. 2013. “Functionalism.” In *The Stanford Encyclopedia of Philosophy*, edited by Edward N. Zalta, Fall 2013. http://plato.stanford.edu/archives/fall2013/entries/functionalism/.
43. Lilly, John Cunningham. 1968. *Programming and Metaprogramming in the Human Biocomputer: Theory and Experiments*. New York: Julian Press.
44. “Lorenz, Konrad.” 2009. *Encyclopedia Britannica Ultimate Reference Suite.* Focus Multimedia Ltd.
45. Losee, Robert M. 1997. “A Discipline Independent Definition of Information.” *Journal of the American Society for Information Science* 48 (3): 254–69. https://doi.org/10.1002/(SICI)1097-4571(199703)48:3<254::AID-ASI6>3.0.CO;2-W.
46. Lu, Chen-Guang. 1999. “A Generalization of Shannon’s Information Theory.” *International Journal Of General System* 28 (6): 453–490.
47. Luckerson, Victor. 2014. “5 Very Smart People Who Think Artificial Intelligence Could Bring the Apocalypse.” *Time*, December 2, 2014. http://time.com/3614349/artificial-intelligence-singularity-stephen-hawking-elon-musk/.
48. Lyons, John. 1977. *Semantics*. Cambridge University Press.
49. Markov, Krassimir, Krassimira Ivanova, and Ilia Mitov. 2007. “Basic Structure of the General Information Theory.”
50. McCarthy, John. 2007. “What Is Artificial Intelligence?” *Stanford University*, November. http://core.kmi.open.ac.uk/download/pdf/6257622.pdf#page=1529.
51. McCorduck, Pamela. 2004. *Machines Who Think: A Personal Inquiry into the History and Prospects of Artificial Intelligence*. 2 edition. Natick, Mass: A K Peters/CRC Press.
52. Moyer, Justin. 2014. “Why Elon Musk Is Scared of Artificial Intelligence — and Terminators.” *The Washington Post*, November 18, 2014. http://www.washingtonpost.com/news/morning-mix/wp/2014/11/18/why-elon-musk-is-scared-of-killer-robots/.
53. Nilsson, Nils. 1998. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann.
54. Oaklander, L. Nathan. 2004. *The Ontology of Time*. Amherst, N.Y.: Prometheus Books.
55. O’Connor, Timothy. 2014. “Free Will.” Edited by Edward N. Zalta. *The Stanford Encyclopedia of Philosophy*. http://plato.stanford.edu/archives/fall2014/entries/freewill/.
56. Oppy, Graham, and David Dowe. 2011. “The Turing Test.” Edited by Edward N. Zalta. *The Stanford Encyclopedia of Philosophy*. http://plato.stanford.edu/archives/spr2011/entries/turing-test/.
57. O’Reilly Media. 2004. “The History of Programming Languages.” 2004. http://archive.oreilly.com/pub/a/oreilly//news/languageposter_0504.html.

58. Parmar, Aarati. 2001. “The Representation of Actions in KM and Cyc.” *Technical Report*, Stanford University, . https://www.researchgate.net/publication/2535883_The_Representation_of_Actions_in_KM_and_Cyc .
59. Poole, David, Alan Mackworth, and Randy Goebel. 1998. *Computational Intelligence: A Logical Approach.* New York: : Oxford University Press.
60. Pratt, Terrence W., and Marvin V. Zelkowitz. 2000. *Programming Languages: Design and Implementation*. 4th edition. Upper Saddle River, NJ: Prentice Hall.
61. “RHUD.” 2002. *Random House Webster Unabridged Dictionary*. Random House Reference.
62. Ritchie, Dennis M. 1993. “The Development of the C Language.” Lucent Technologies Inc. 1993. https://www.bell-labs.com/usr/dmr/www/chist.html.
63. Russell, Stuart J., and Peter Norvig. 1995. *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Intelligence. Englewood Cliffs, N.J: Prentice Hall.
64. ———. 2010. *Artificial Intelligence: A Modern Approach*. 0003 ed. Upper Saddle River: Prentice Hall.
65. Sammet, Jean E. 1969. *Programming Languages: History and Fundamentals*. Prentice-Hall.
66. Schaeffer, Jonathan. 2009. *One Jump Ahead:: Challenging Human Supremacy in Checkers*.
67. Schmitt, W. F. 1988. “The UNIVAC SHORT CODE.” *Annals of the History of Computing* 10 (1): 7–18. https://doi.org/10.1109/MAHC.1988.10004.
68. Searle, John R. 1980. “Minds, Brains, and Programs.” *Behavioral and Brain Sciences* 3 (03): 417–424. https://doi.org/10.1017/S0140525X00005756.
69. ———. 1990. “Is the Brain’s Mind a Computer Program?” *Scientific American* 262 (1): 26–31.
70. Sebesta, Robert W. 1993. *Concepts of Programming Languages*. 2nd ed. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc.
71. Shannon, Claude E. 1950. “Programming a Computer for Playing Chess.” *Philsophical Magazine*, 7, 41 (314). http://vision.unipv.it/IA1/ProgrammingaComputerforPlayingChess.pdf.
72. Shannon, Claude E., and Warren Weaver. 1948. “A Mathematical Theory of Communication.” *Bell System Technical Journal* 27: 379–423, 623–656.
73. Shannon, Claude Elwood, and Warren Weaver. 1963. *The Mathematical Theory of Communication*. Urbana: University of Illinois Press.
74. Simon, H.A. 1965. *The Shape of Automation for Men and Management, New York: Harper & Row.*
75. ———. 1969. *The Sciences of the Artificial*. First edition. Cambridge: MIT Press.
76. Smith, Michael, Chris Welty, and Deborah McGuiness. 2004. “OWL Web Ontology Language Guide.” W3C. 2004. https://www.w3.org/TR/2004/REC-owl-guide-20040210/.
77. Sowa, John F. 2000. *Knowledge Representation: Logical, Philosophical and Computational Foundations*. Pacific Grove, CA, USA: Brooks/Cole Publishing Co.
78. ———. 2010. “Ontology.” KR Ontology. 2010. http://www.jfsowa.com/ontology/index.htm.
79. Stonier, Tom. 1990. *Information and the Internal Structure of the Universe: An Exploration into Information Physics*. London: Springer-Verlag. //www.springer.com/gp/book/9783540198789.
80. Strevens, Peter, and F. F. Weeks. 1984. *Seaspeak Reference Manual*. Pergamon Press.
81. Sunik, Boris. 2003. “The Paradigm of OC++.” *SIGPLAN Notices* 6: 50–59.
82. ———. 2013. “Theory of Meaningful Information, Background, Excerpts and Meaning Representation.” *Artificial Intelligence Research* 2 (3): 102–22. https://doi.org/10.5430/air.v2n3p102.
83. Weizenbaum, Joseph. 1976. *Computer Power and Human Reason: From Judgment to Calculation*. San Francisco: W H Freeman & Co.
84. “What Is Meta? - Definition from WhatIs.Com.” n.d. SearchSQLServer. Accessed March 8, 2018. http://searchsqlserver.techtarget.com/definition/meta.

85. Yule, George. 2006. *The Study of Language*. Cambridge University Press.