How to Do Things with Keys

(Assembly) Programming as (a Kind of) Gesture

Stefan Höltgen (Berlin)

I. Introduction: Flusser, Computers, and Programming

A first glance at Vilém Flusser's writings reveals that he perceives the computer as a very dubious medium which negatively affects the individual: "Through the codes of technical pictures the masses become illiterate," he writes in *The Gesture of Writing*, continuing that "the system analyst does not need to write because the computer works without an alphabet [...]" (Flusser 1994: 39 – own translation) And: "But creation is a term defined by computer studies, thanks to cybernetic apparatuses, and one can see that the machine is far more creative than any human can be, if it is programmed by a human or another machine to be creative." (Flusser 1994: 28–29). I have cited these quotes for two considerations that are following Flusser, criticize, and specify him. In the end, I want to define his particular use of the word "programming" – a term which he uses very often[1] in his writings.

It should be emphasized that the intention is not to criticize Flusser's use of technical terms (cf. Sokal & Bricmont 1998). Rather, it is merely to show that Flusser's use of the word "programming" opens a relevant epistemological discourse. He is not a "technophobe" or a "technical illiterate" when he uses technical terms as metaphors. In *The Alphanumeric Society* (1987) and *The Emigration of Numbers from the Alphanumerical Code* (1991), Flusser described language processing automatons according to their main features:

> "We do not all need to write numbers anymore; neither do we have to read them because these practices have become beneath us. Now it is our duty to manipulate the structure of the "universe of numbers" (i.e., to program the machines for their calculations). This step back from counting to analyzing and synthesizing structures opens the door for formal thinking. Such thinking must become code to be articulated. [...] It has been shown that computers do not only calculate; surprisingly, they also compute. They do not only fragment algorithms into numbers (into bits), they also collect those bits and shape them into e.g. lines, areas, bodies, animations, and sounds." (Flusser 1999: 52 – own translation)

Without its techno-critical subtext, this quote shows that Flusser knows very well what programming is about: setting up a machine for the formal handling of numbers: building algorithms that can be used from the machine to manipulate the "alphabet of numbers" and to calculate any task which is calculable.

II. Programming Between Flusser's Gestures of "Making" and "Writing"

The activities of the programmer are a very special craft. Programming resembles writing – a gesture Flusser addresses in a chapter of his book. He defines writing as an act which can be used to let "thoughts become reality in the form of a text" (Flusser 1994: 29). For this purpose, the writer needs special materials, rules, and ideas. (cf. Flusser 1994: 33) The gesture of writing follows a "specific linearity" (Flusser 1994: 33), especially when a typewriter is used to write. Using a word processor (which Flusser discusses as a special form[2] of typewriting [cf. Flusser 1994: 35, 59]) suspends this linearity because then there are two texts working together: the "surface text" which is written and edited by the user and the "subtext" which rules the writing and editing of the surface. This second text is the program code. It has to be written, too, and in practice it is also written with computers – which lets it become a recursion of itself. The programming language text on the subtext has a special purpose as a communication tool and follows certain rules, both of which distinguish it from the surface text.

Programming is indeed a kind of writing in the manual sense. But its text (the program code) has to be saved and transferred and also has to become operative (to run the algorithms). So it is perhaps more related to another kind of gesture: making. According to Flusser's theory, "making" involves the use of both hands in a dialectical movement to inform, transform, or move matter with power, drive, and in cooperation (of both hands) (cf. Flusser 1994: 49). Especially if there is a physical transformation, making becomes the most obvious of Flusser's gestures. I will analyze a specific programming language that carries the idea of "making" in its name: assembly language, which affects the computer hardware directly by making things.

Next I approach the gestures of "writing" and "making" from the views of computer science and linguistics. This will show the kind of writing and alphabet we are handling and how the programmer and the computer put writing into operation in order to do something with it.

III. Do You "Write" a Computer Program?

The Theoretical Perspective: Formal Languages and Automatons

From a theoretical perspective, computers are built from text. For this reason, it could be useful for the purposes of understanding gesture to present this theory in detail. Theoretical computer science considers the question of how automatons can handle (analyze and synthesize) formal languages. To do this, it uses a definition of language which is significantly broader than the one Flusser uses:

"An *alphabet* is an arbitrary nonempty finite set of symbols equivalent to the script of a natural language. The term *word* over an alphabet corresponds to any text that is comprised of symbols of this alphabet. Any set of words (texts) over the same alphabet is called a *language*." (Hromkovic 1998: 51)

Language is defined as a set of *symbols* (or strings of symbols) and words. These *words*[3] are built out of the symbols on syntactic rules. By using *grammar*, a language can be built out of words. Looking at it from the opposite perspective: words that have been built with a grammar are elements of a language that uses this grammar. But grammars are not the only way to analyze (parse) or build (derive) words: automatons can manage this, too. This is how computers can recognize languages and languages can put the computer into operation.

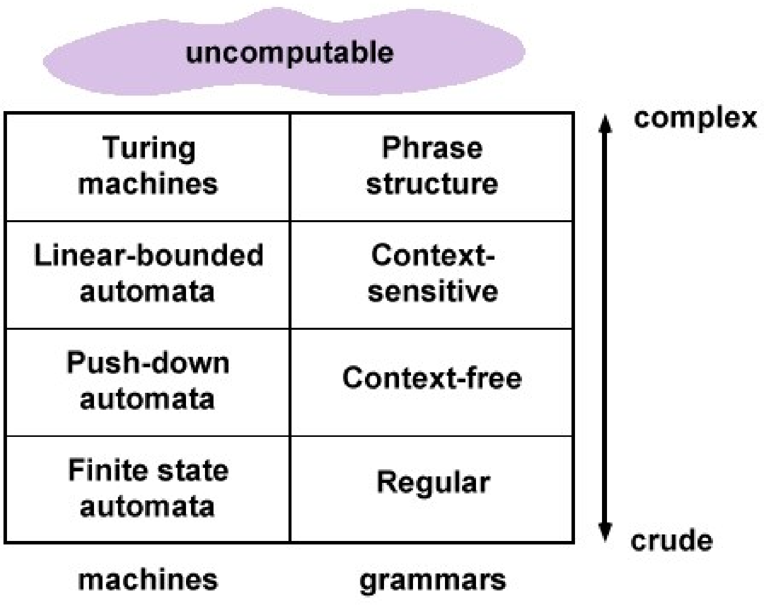


Fig. 1: The Chomsky hierarchy

Thus Flusser's argument that "computers work without an alphabet" seems only legitimate from a very anthropomorphic point of view of language. De facto computers can build and understand very complex alphabets. In 1956, Noam Chomsky published his theory on the types of formal languages in which the complexity of those languages is linked to the automatons that can handle them. He distinguishes between regular languages (for finite-state machines), context-free languages (for pushdown automatons), context-sensitive languages (for linear bounded, nondeterministic Turing machines), and formal grammar languages (for common Turing machines). These theoretical automatons (cf. Martin 1972: 422–425) can be implemented as real machines, in which case we are speaking of computers. Computers as they are commonly used today are implemented common Turing machines. They can work with formal languages that are built from unlimited grammars. This means that the computer recognizes every word of a specific language, but it cannot decide if a word that it does not recognize is an element of that language.

This is the definition of the “halting problem”. While writing a program, lots of words have to be recognized that afterward must be analyzed semantically in order to assign them to specific operations. This is usually not a problem, as long as these words are elements of the instruction set. But there are words such as functions, variables, and data that have to be recognized, too. These words are used for complex tasks in iterations and recursive functions. This is the origin of the halting problem, because the computer cannot decide if a specific function leads to one result or to no result. It has to compute it with brute force, which literally can last forever. It falls to the programmer to define break conditions for loops to avoid the halting problem.

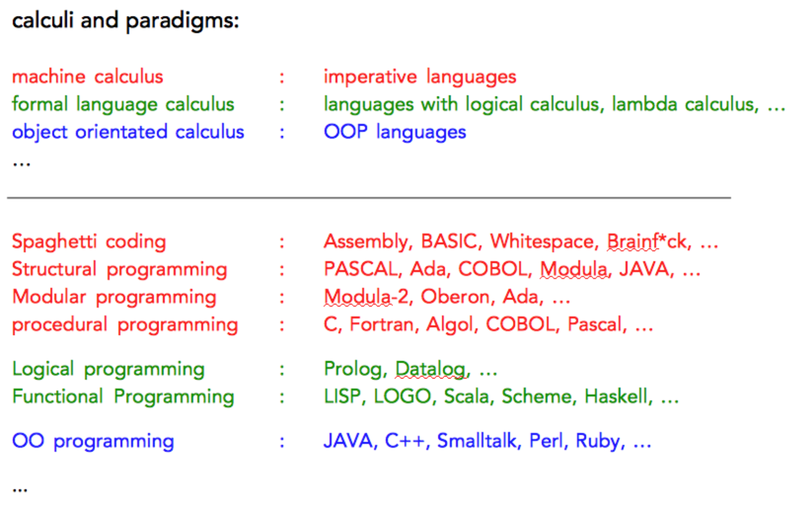
The Practical Perspective: Calculi and Paradigms

That leads us to the practical perspective. Special programming languages have been developed over the last 70 years of programming practice. The number of programming languages is neither manageable nor presentable. But languages can be classified according to: a) how they formalize a task to be solved by a computer (the calculus) and b) how they are used in a stylistic manner (the paradigm they are programmed with).

Programming languages can be distinguished by their calculi. Mainly three calculi can be observed:

1. Languages that follow the machine calculus: imperative languages. Their programs contain a sequence of instructions.
2. Languages that follow formal language calculi: e.g., the logical calculus or a mathematical (lambda) calculus in which the tasks are described by the programmer. They are called declarative languages.
3. Object-oriented languages: these language try to express tasks as descriptions of objects and their relations to each other (which follows a kind of cognitive scientific approach).

The programming paradigms can be assigned to each of these calculi. The paradigms grew historically, complemented or replaced each other. Structural, modular, procedural paradigms are mainly programmed with imperative languages. Logical or functional paradigms are mainly programmed with declarative languages. While the first classification is “hard” (every language is assigned to one specific calculus) the second classification is “soft” (you can program any paradigm with any language).


calculi and paradigms:

machine calculus	:	imperative languages
formal language calculus	:	languages with logical calculus, lambda calculus, …
object orientated calculus	:	OOP languages
…		

Spaghetti coding	:	Assembly, BASIC, Whitespace, Brainf*ck, …
Structural programming	:	PASCAL, Ada, COBOL, Modula, JAVA, …
Modular programming	:	Modula-2, Oberon, Ada, …
procedural programming	:	C, Fortran, Algol, COBOL, Pascal, …
Logical programming	:	Prolog, Datalog, …
Functional Programming	:	LISP, LOGO, Scala, Scheme, Haskell, …
OO programming	:	JAVA, C++, Smalltalk, Perl, Ruby, …
…		

Fig 2: Calculi and Paradigms

The cause for this "soft" classification is that the only "real" language a computer can operate with is a machine language. Any higher programming language that follows a calculus other than a machine language must be compiled into and interpreted as a machine language in order to run on a computer. Of course the programmer can decide to use a machine language in the first place; doing so has some advantages (higher performance, dense algorithms) and some disadvantages (the size of modern programming projects, the poor clarity of the code). But of course it is possible to make a program structural, modular, procedural, logical, functional, and object-oriented with machine language, too. It only depends on the style[4] in which the programmer writes the source code. This source code is the writing in which the programmer expresses tasks as algorithms. To identify this writing as a special kind of text would not solve the problem because the purpose of a computer program is not to be written or to be read but to be executed.

IV. What Does the Assembly Programmer "Do"?

I focus on the execution of a machine language program because it is the language with which the computer "makes" something. Its alphabet contains only two symbols: "0" and "1". Any word that can be built from these two symbols with distinctive rules is an element of this machine language. The grammar of this language defines elements such as microcodes (the "morphemes" of an instruction that activate the electronic gates), the length of the words (which depends on the width of the processor buses), the addressing modes (which define how to access memory for loading and writing data), and so on.

Here is an example:

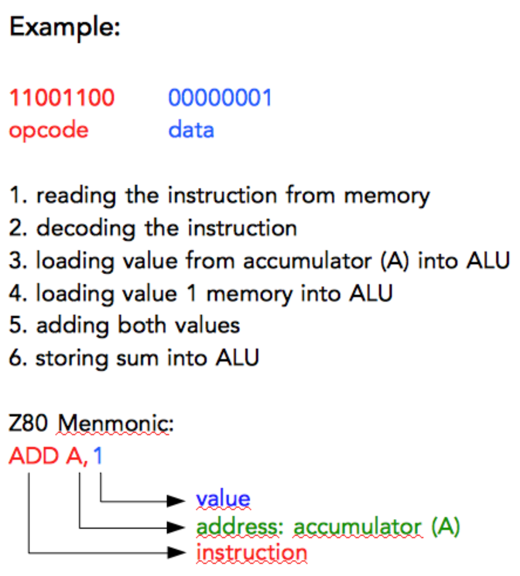

Example:
11001100
opcode
00000001
data
1. reading the instruction from memory
2. decoding the instruction
3. loading value from accumulator (A) into ALU
4. loading value 1 memory into ALU
5. adding both values
6. storing sum into ALU
Z80 Menmonic:
ADD A, 1
value
address: accumulator (A)
instruction

Fig 3: Example

These two words are elements of the machine language of the Z80 microprocessor. The left 8 bit group contains the opcode (the actual instruction); the right 8 bit group contains data. With this specific instruction-data combination, the computer adds the number "1" to the content of a memory called the "accumulator" (A).[5] Executing this instruction starts several work steps: in the decoding phase, the instruction will be recognized and specific electronic gates will be activated. In the execution phase, the "old" value from the accumulator and the "new" value "1" will be transported to the gates where they are added. Then the sum will be transported back to the accumulator. (Cf. Zaks 1981: 55–61, 200)

This example shows the intricate structure of machine language. Its words are stored within the microprocessor. While decoding, the instruction will be compared to this hard-wired lexicon. On this level, the computer reveals its nature as the implementation of a (theoretical) Turing machine. The binary symbols are its alphabet. They are building the language this Turing machine recognizes. For humans, the binary presentation (that itself is a symbolization of hardware switching states) is hard to read and to understand. One of the first challenges for 1940s computer engineers was to build a language which could be understood more easily. The invention of a mnemonic assembly language for those opcodes was the result. Assemblers were programmed to translate them back into machine language. They use the same grammar as the machine language, only utilizing another, more human, alphabet for the opcodes. The instruction above can be written as "ADD A,1" in this assembly language.

The Instruction Set of the Microprocessor as a Lexicon

Assembly and machine languages are imperative languages. Computers are built to read and execute programs as a set of instructions. In order to do this, the instructions have to be defined distinctively. Formal languages cannot be deconstructed. In contrast to natural languages, formal languages do not have any ambiguities and cannot be "interpreted" with varying results. The relation between the signifier and the signified is not arbitrary but clear, in the form of the logic gates that are "meant" by the "morphemes" of the instruction. Every machine language program generates a special Turing machine (to solve a specific task) which means: there is a specific electronic circuit for this task which does the same thing in reality as the program that describes it.

This carries consequences for the programmer of this machine. The "understanding" of a program within the machine becomes apparent in the circuitry by using a logic analyzer; the "intention" of the programmer can be calculated with mathematical functions. Any errors (apart from hardware design flaws[6]) are the result of the incorrect usage of the programming language by the programmer: using wrong words that are not part of the language (syntax errors), semantically badly constructed algorithms (logical errors), or poorly translated aspects of the "real task" into the formal language (runtime errors).

It is only possible to translate those real world tasks into machine language that are computable. The amount of "computable tasks" is so very huge and complex that even big problems that are considered to be "not computable" become computable after they are analyzed and reduced to a set of smaller computable tasks (cf. Hromkovic 1998: 221–222).[7] In other words: for plenty of tasks, it is possible to find a machine language program that solves them. Through the increase of memory, the acceleration of process cycles, and the infinite possible combinations of words (from the formal language), the borders of computability are expanding continuously.

Programming as a (Speech) Act

Can it thus be derived from the character of imperative programming that programming assembly is a kind of speech act? The "speech act theory" introduced by John L. Austin (Austin 1962) and expanded by John R. Searle (Searle 1969) tries to present language not only as a communication system but as a set of tools. With language, it is possible to determine someone (or yourself) for an act, state a fact, and express a feeling. Speech act theory analyzes the performative aspects of natural languages, but is it possible to extend this to formal languages? Specifically: what is the *particular action* the programmer performs while programming a computer?

Searle defines the contents of speech acts that are trying to affect the listener to do something as *directive illocutions* or more briefly as *directives* (Rolf 1990: 185). One can assume that programming is a speech act because the instructions of the imperative programming paradigm in many programming languages are expressed with imperative verb forms: PRINT, DO, REPEAT, OPEN, et cetera in high level languages; or ADD (add), CMP (compare), JMP (jump), LD (load), and so on in assembly languages. The performative impact is revealed in the *temporal relation* to its effect on the listener: at once or shortly after the instruction was given, an effect occurs that has a causal correlation to the former speech act. The execution of a programming instruction in the computer only occurs after the compilation/interpretation into machine language and at run time.[8] Only then are the program instructions loaded from the memory to be decoded and executed. So the causal nexus and the temporal relation Searle demands for speech acts exist – but only between program fetch (perlocutionary act) and program execution (perlocutionary effect), and not between programming and program execution. As long as they are not compiled *and* executed, computer programs are only binaries and source codes or, as Friedrich Kittler says, "literature for compilers" that remains to be read...

From this point of view,[9] programming cannot be a speech act between the programmer and the computer because it is the computer itself that fetches the instructions from memory to execute them. This is why the engineers of the 1940s called programming "automatic coding". The user only activates that process by loading and starting the program. And this user need not be the programmer; we all know this because we use programs that we did not program ourselves. The temporal "delay" between programming and executing forms the basis of the software industry and the usage of the computer as a tool.

V. Conclusion: People Programming Computers Programming People

Writing source code differs from writing a novel: programming in the sense of theoretical computer science means building a paper machine – a theoretical machine that solves a set of distinctive computational tasks. With programming, this process happens solely with symbols and with the power of thoughts. The programmer is in a mental state where it is possible to think the functions of the program in order to find a proper paper machine for the task. Assembly programmers cannot access abstract formalisms to do this – they must perform the functions of the machine mentally. This is much more than the "gesture of writing" that Flusser describes as "a phenomenalizing of thoughts" (Flusser 1994: 35). It is a symbolizing of the operating machine.

Programmers "do" very little in a concrete sense of the word. At most, they press keys – a gesture that physically cannot be distinguished from typing text with a word processor.[10] The actual activity does not take place in the programmer's hands but in the brain. But according to Flusser, the gesture of making also implies intellectual activities (Flusser 1994: 49). This is part of the definition of programming because in the process of writing source code, the assembly programmer projects and rejects algorithms, tests routines, searches for and fixes bugs – mentally building/ making/ assembling a paper machine. In doing this, the programmer is in close contact with the machine. Concentration is necessary because the calculus and the paradigms of the machine's language differ in so many aspects from the programmer's.

The evolution of higher programming languages is based on the idea of "humanizing" this process by implementing concepts of natural languages or other calculi than the machine calculus.[11] Only by doing so can the intimate process of programming be alienated (in the Marxist sense) and the idiosyncratic communication between the programmer and the computer can be abolished to transfer the act of programming into an economic (in the Marxist sense, too) process.[12] For economic reasons, assembly languages are programmed today only by developers of higher programming languages. So it is possible to say that assembly programming abolishes itself.

Outside of formal languages, gestures are not clear (cf. Flusser 1994: 7–8). They must be interpreted and they can be deconstructed. This is precisely the reason for the aforementioned problems with defining programming as a "gesture of writing" or a "gesture of making". Rather than being one or the other, programming contains aspects of both gestures: while writing source code, the programmer creates his own idea of a Turing machine, he is "programming and transcending" (cf. Flusser 1994: 7–8) himself by using ("speaking") a programming language to think like the machine. Programming additionally contains aspects of "making". Flusser argues that there is a connection between the mental and manual actions that are stored in the meanings of words like handle, uptaking, bearing (Flusser 1994: 50), and of course assembling.

Programming shifts between writing and making – it is unable to be only one without being the other, too. This coincides with Flusser's use of the word "programming" as a metaphor. Strikingly, he often argues that people are not programming machines, but instead machines are programming people. In the introduction of his *Gestures* book, he advises against using technical terms in humanities discourses to define artistic, ethical, or other "interpretable problems" (cf. Flusser 1994: 9) of the humanities with causalities. If we do not want to accuse him of self-contradiction, we can assume he is attempting to think as a cybernetic theorist. Cybernetics is the theory of control and communication in animals and machines because both systems can be described in terms of programming – as being programmed. The main differences are that machines are operating while people are performing, and that the ability of animals and humans to disobey an instruction and to not perform reflects a choice the machines do not have. Programming in the sense of operating is neither a gesture nor a "cultural technique". When Flusser blurs the difference between operation and performance by using the term "programming", he questions our belief in intellectual self-determination while simultaneously writing science fiction about artificial intelligence.

Stefan Höltgen – PhD (1971) – studied Linguistics, Philosophy, Social Sciences and Media Studies from 1996 to 2000 in Jena. 2009 he did a dissertation in German Literature Studies in Bonn about „Discourses of Media and Violence in Authentic Serial Killer Movies". 2008 he moved to Berlin where he is working as publicist. In 2011 he started a post-doc research on „The Archaeology of Early Micro Computers and their Programming" at the Department of Musicology and Media Studies (Berlin Humboldt University) which is also a second dissertation at the Center for Computer Studies (Berlin Humboldt University). He publishes and edits books and papers on the history of computing, computer games and the archaeology and epistemology of media as well as reviews of video games, movies, and books. His key subjects are computers, media, technology and their history. In 2015 he started a book series on computer archaeology and joined the editorial staff of the magazine „Grundlagenstudien in Kybernetik and Geisteswissenschaft" (Fundamentals in Cybernetics and Humanities). He curates the annual „Vintage Computing Festival Berlin".

www.stefan-hoeltgen.de; e-mail: stefan@hoeltgen.org

1 The search for the term “program*” in Flusser's writings returns hundreds of results. Such a quantitative search is not an end in itself but shows a special case of the theories of formal languages since “regular expressions” (like the asterisk) are used to find terms. The intent is to show that Flusser does not use the terms “program” and “programming” primarily as metaphors.

2 “Undoubtedly the computer radically changed the writing process and the attitude of the writer and the recipient of texts. The experience of creative engagement has changed. A new kind of self criticism and responsibility for the reader has occurred and the text takes on a kind of life of its own. In a sentence: If you write this way, you will think, create, and live in dialogues.” (Flusser 1999: 66 – own translation)

3 “Words” are all kinds of signs (and groups of signs) that can be understand as the contents of a language. They can be individual signs (letters, ciphers, and other symbols) or groups (tokens) of them.

4 For the term “stylus” of programming, cf. Hagen 1992 and Hagen 1997.

5 According to Flusser, computers reduce every calculation to additions (cf. Flusser 1999: 208). This is only a metaphor for the immense simplification of programmed real world problems down to the machine language level. The instruction set of microprocessors was larger than 1000 opcodes in the mid-1970s – opcodes for addition were just a small part of that. Even on the electronic level of the microcodes, adders are used for more than adding, though there are far more functions that are using more and other gates than the adding gates.

6 The Pentium bug is an example of this.

7 This leads to the question of whether “big data” analysis can solve problems faster and better than theory driven science can. (Cf. Mainzer 2014)

8 The only exception to this are imperative languages with a command line interface (CLI) which executes every command right after the input. These languages are irrelevant for (longer) programming projects and are only used for process control.

9 There are other objections: a machine-given instruction (there are performative speech acts between machines and humans: car navigation systems are an example) can be refused by the human recipient because of free will. The computer lacks this freedom when running a program. For Austin and Searle, language contains a performative moment. “The distinction between operative and performative defines a mode of information processing – regardless of whether any consciousness is involved in this process.” (Heiseler 2013 – own translation) A performative speech act can be refused performance by its receiver. This “loose” connection between speech act and speech effect has been a topic of debate between Searle and Derrida. For post-structuralist linguistics, the lack of references of the signifiers are the starting point for the deconstruction of meaning. Austin and Searle did not analyze the signification process but the perlocutionary act of speaking, which has an agent (the speaker) and a referee (the receiver). To presuppose an effect of the speech act would lead to a “fixed” language. Derrida rejects this as a kind of psychologism. From the post-structuralist point of view, formal languages are not deconstructable. They would be if they were performative speech acts.

10 As a matter of fact, programs are often written with text editors. For this reason, their impact as tools for thinking and writing must not be underestimated.

11 The speech act theory itself contains the capability of being a formal language: John McCarthy published a programming language named “Elephant 2000” in 1998 which implements the speech act theory as a programming language. “Elephant 2000” does not try to formalize natural language but to show programmable concepts of interaction. (Cf. McCarthy 1998). Beyond this, the speech act theory is used in the research of artificial intelligence. (Cf. Cohen/Perrault 1981)

12 “The majority of people do not work for themselves. They serve as tools for the working ones.” (Flusser 1994: 20 – own translation) “In a logical sense the man is an attribute of the apparatus because he can be exchanged by another man while working.” (Flusser 1994: 27 – own translation). This can be transferred to the situation of the programmer which becomes a tool for the machine (that get its source code from his work to transform it into binary code). If the source code is written in assembly language it is almost impossible to exchange the programmer because his particular imagination of the machine leads to very

idiosyncratic ideas and algorithms. Higher programming languages – especially object oriented languages – are used to get a division of work within the programming process – which means: to make the programmer exchangeable.

Cited literature

Austin, John L. Zur Theorie der Sprechakte (How to do things with Words). Stuttgart 1994.

Cohen, Philip R., Perrault, C. Raymon. “Elements of a Plan-Based Theory of Speech Acts.” In: Lynn Webber, Nils J. Nilsson (eds.), *Readings in Artificial Intelligence.* Los Altos 1981: 478–495.

Flusser, Vilém. “Die Auswanderung der Zahlen aus dem alphanumerischen Code.” In: Dirk Matejovsk, Friedrich Kittler (eds.), *Literatur im Informationszeitalter.* Frankfurt am Main/New York: Campus, 1994: 9–14.

Flusser, Vilém. *Gesten. Versuch einer Phänomenologie.* Frankfurt am Main 1994.

Flusser, Vilém. *Medienkultur.* Frankfurt am Main 1999.

Hagen, Wolfgang. “St(ab)il – Instabil Über den Zirkel des Programmierens.” In: Forschungsinstitut für Anwendungsorientierte Wissensverarbeitung (ed.), *Technik, Öffentlichkeit und Verantwortung.* Ulm 1992: 127–173. (also: http://txt3.de/flussergestures1 . 23.02.2015.)

Hagen, Wolfgang. “Der Stil der Sourcen.” In: Wolfgang Coy, Georg Chr. Tholen, Martin Warnke (eds.), *Hyperkult. Geschichte, Theorie und Kontext digitaler Medien.* Basel u.a.: Stroemfeld, 1997: 33–68. (also: http://txt3.de/flussergestures2 . 23.02.2015).

Hromkovic, Juraj. *Theoretical Computer Science. Introduction to Automata, Computability, Complexity, Algorithmics, Randomization, Communication, and Cryptography.* Berlin 2004.

Mainzer, Klaus. *Die Berechnung der Welt: Von der Weltformel zu Big Data.* München 2014.

Martin, David F. “Formal Languages and Their Related Automata.” In: Alfonso Cardenas, Presser, F., Marin Leon, Miguel A. (eds.), *Computer Science*. New York 1972: 409–460.

McCarthy, John. “Elephant 2000. A Programming Language Based on Speech Acts.” In: http://txt3.de/flussergestures3. 23.02.2015.

Rolf, Eckard. “On the concept of action in illocutionary logic.” In: Armin Burkhardt (ed.), *Speech Acts, Meaning and Intentions. Critical Approaches to the Philosophy of John R. Searle.* Berlin/New York 1990: 147–168.

Searle, John R. *Sprechakte. Ein sprachphilosophischer Essay.* Frankfurt am Main 1983.

Sokal, Alan, Bricmont, Jean. *Fashionable Nonsense. Postmodern Intellectuals' Abuse of Science.* New York 1998.

von Heiseler, Till Nikolaus. *Operativer Code [Vorläufige Notizen].* http://txt3.de/flussergestures4. 23.02.2013.

Zaks, Rodnay. *How to program the Z80.* Düsseldorf 1981.