Perceiving Computers

- A comparative armchair analysis of how language influences the way ordinary computer users and computer experts perceive computers.

Andreas Lloyd

2005

Introduction

Within the past 30 years, the computer has become a metonymy for technology – an integral part of basically any new technological development produced, so pervasive in our everyday life that we're hardly even aware of its presence at all. But even so, computer technology is running our phones, our kitchen equipment, our cameras, stereos and video players. It is used to organise timetables and money transactions, taxes, criminal and health records, surveillance, vote counting – all of these functions beyond what we usually associate with the personal computer, in itself a tool so incredibly versatile that most of us are only aware of a fraction of its potential.
Within the last 15 years or so, the personal computer has become a basic necessity for all white collar work, opening up new avenues of digital communication such as email, web pages and instant messaging, even for managing photos, videos and music collections.
All of a sudden, everybody has had to become computer *literates*, though this transition has not been easy for everyone. The introduction of the computer as a universal information tool has instituted changes in the routines of work and everyday life in such a massive fashion that we seem to forget how quickly we are getting to take these changes for granted. Only few people have actually gotten to terms with the machine they use and rely on for all their office work, and – to an increasing degree – their communication as well.

This gap in ability has been spanned by the computer experts, the curious few whose pure enthusiasm for the new technology originally opened the market for the first personal computers (cf. Levy 1994). These people are maintaining the velocity of modern society in its already now extreme dependence on computing technology through their knowledge of the abstract, intangible nature of software and the inner workings of the computer. The way that these experts perceive computers is so radically different from the average user that most tend to ignore it, yet when we hear them talk with each other about these machines that we rely on, and that we rely on these experts to keep working, it is hard to ignore this specialized language of metaphors and acronyms which computer experts easily and fluently. Sentences such as "Open port 22 in the firewall to allow the SSH-client access" indicate an esoteric language that both directly and indirectly may be structuring experts' and users' perception of computers.

In this essay, I will partly use my own observations and experiences as a computer supporter, partly the limited literature[1] available on the use and perception of computers in a sort of "armchair" ethnographic analysis comparing how the average everyday computer users and self-professed computer experts perceive and conceptualize the computer. I will use this comparison to explore how language may be structuring our perception of computers, and as an example of the ever-ongoing discussion of whether language in general is determining the ways in which we think and perceive the world.

[1] The perception and use of computers has not drawn much attention in its own right within the rather technology-shy discipline of anthropology. The limited anthropological focus has so far been on the new possibilities for communication and social interaction opened by the rise of computers and telecommunications, rather than the varying social and cultural perception and use of computers inherent in the technology in its own right. Though anthropologists are now beginning to recognize the importance of examining the perception and understanding of computers, the groundwork of this field has long since been developed by psychologists such as Sherry Turkle, computer scientists such as Gerald Weinberg and even popular science writers like Steven Levy. This literature is not based on any direct ethnographic fieldwork, but can rather be compared to the accounts of colonial engineers, soldiers and missionaries upon which early anthropologists for most of the empirical data. As the field of computers is still so new, I've found it necessary to base this essay on "armchair" sources.

The ordinary computer user

Having worked in a university computer support department for several years, I have had the opportunity to observe ordinary users' computer problems firsthand. Many users have only been using computers for a few years, and often their introduction to the technology has been helped to a great extent by the Graphical User Interface (GUI) based on the "desktop" metaphor and popularised by Microsoft Windows through the 1990s.[2] The desktop metaphor should be familiar to almost any computer-literate person today,[3] yet, it is mere gloss upon the true, and much more complicated nature of the computer, consisting of complex layers of abstract, mathematical symbols with little apparent connection to the functionalities known by the ordinary user. Therefore, there is often little connection between the problem a user may be having, and the relevant solution.

In his article "Spirits as 'ready to hand'" (2004), the Danish anthropologist Rane Willerslev argues that we most often use 'recipe knowledge' to assume mastery of the technology we use, without necessarily understanding the inner workings of these technologies. This allows us to use a technology without paying it any attention *as such*, it becomes a transparent tool to fulfill our work – such as using a computer to write an essay but not noticing the computer, merely the work being written. This is what Heidegger calls 'ready-to-hand' – an ease of use that removes conscious focus from the tool onto the work itself.

In the case of computers, it is through the GUI, the computer becomes 'ready-to-hand' for the average user who, with his 'recipe knowledge' of that GUI, can use it as tool without further thought as to *how* it works.

But the moment that the tool no longer functions as expected, as it becomes 'unready-to-hand', the shock of its failure brings the tool back into the consciousness of its user, and awakes a need for negotiation and solving of this new problem (Willerslev 2004: 401-405). Willerslev uses this to explain the Yukaghir hunters' relationship to the spirits whom they offer gifts in order to secure success on the hunt. As one hunter

[2] For more on the history of Graphical User Interfaces and the Desktop metaphor, see http://arstechnica.com/articles/paedia/gui.ars

[3] In the terminology of the "Desktop" metaphor, the screen is virtual desktop, displaying the computer's file system as an intricate hierarchical system of folders and files. There is a virtual trash bin where unwanted files can be dragged and deleted, and there are various applications with the look and feel of other, familiar technologies. I.e. a word processor is basically the same as a typewriter, a media player plays cds and DVDs much like an ordinary DVD player, etc.

explains it:

It is like the way in which you use your computer. You write on it, but you do not think about how it actually works. You have told me this yourself. You just need to work on it, not to understand how it works. It is the same with me. I need to do so and so to kill an elk, but I do not think about its deeper meaning and I do not need to know.
(Willerslev 2004: 402)

When the hunt fails, and no elks are to be found, the shock will force the Yukaghir hunters to negotiate a solution with the spirits, for instance by figuring out that one hunter had upset the local spirit by killing a pregnant dog and quickly proceeded to exclude him from the hunt (Ibid. 406). Like the Yukaghir hunters, who have basic prototypical ideas of what the spirits are like (Ibid. 407), most computer users usually have a some notions of what computers are capable of, and what might go wrong (such as vira, spam, “hackers” etc.), so that they easily can generate an explanation for their computer's sudden 'unreadiness-to-hand'. But since many users lack the technical knowledge and vocabulary necessary to give them a clear idea of how to solve it, they are left in need of help and explanation whenever a computer requires expertise of the inner workings of the computer – beyond the limited ‘recipe knowledge’ they possess of the GUI. In these cases, many users often revert to an – often *consciously wrong* – explanation saying that the machine “ill” in order to communicate their computer troubles.
In her book “The Second Self” (1984), the American psychologist Sherry Turkle argues that most computer users from time to time anthropomorphizes the computer, constructing it as a psychological subject that, though based upon a collection of abstract concepts gathered in a core rational, mathematical logic, still can be blamed for mistakes, treated nicely to produce certain results, and be “put to bed” to recover from a nasty virus. Turkle lists a wide range of examples of computer users relating to their machine as more than a mere tool, but rather an independent entity with which negotiations often must be held in order to achieve the desired result. Turkle cites an example of an airline ticket agent whose apology to customers bumped back from overbooked flights is that the computer fouled up (because it has been programmed on the assumption that some passengers won't show for any given flight), rather than laying blame on her coworkers, company policy or some other independent agent

(Ibid. 20-22, 271-274).
Much like in the case of the Yukaghir spirits, the need to consciously reflect on, and anthropomorphize the computer usually arises only in situations when computer does something so unexpected, so outside of the usual pattern of daily use, that it requires an explanation.
In general, there are three main categories of computer problems encountered by the average user, which may trigger such a reaction: Technical problems, user problems and inexplicable problems.
Technical problems are usually straight-forward, though not always easily solvable centered around the computer itself, such as setting a machine up to connect to the internet, or to automatically make a backup every week. User problems usually do not so much require understanding the problem on the level of the machine as it requires understanding what it is the user wants to achieve and helping her towards that goal, for example by showing her functions she wasn't aware of. Inexplicable problems are situations that seem truly inexplicable – such as when opening two specific applications in a certain way and order makes the computer crash. This leaves the supporter to explain the apparently inexplicable, and it is often easier to support the user's already active idea or semi-superstition that the computer is indeed capable of being ill or unhappy from time to time, rather than attempt an explanation necessarily using an amount of computer jargon.
In all cases of computer problems, the authority and expertise of a computer expert is often tied directly to the language he uses. Much like the authority of doctor is tied to his use of Latin denominations of various parts of the body or a mechanic with his specific knowledge denominating the inner workings of a car, the computer expert uses acronyms and conceptual metaphors such as HTML, RAM, Browser or Motherboard to define different parts of the computer.
But unlike repairing a car, where the mechanic can point triumphantly to some clogged-up filter or worn-out ball-bearing; or diagnosing a patient, where the doctor can point to any part of the body, explaining what the Latin name signifies; the computer expert can only rarely solve a problem by opening up the machine and poking at its innards. Most of his vocabulary covers abstract concepts of software bound in esoteric acronyms and vague metaphors with such as firewalls, ports, files

and folders with no apparent physical referents.
When a computer expert is called to solve a problem, he will begin typing away at the keyboard, producing strange results and finally, without anything seemingly having changed much at all, triumphantly claim that the problem is solved. In order to explain to the ordinary user how a problem occurred and how to avoid it, the supporter can say something like “Your text is ASCII-coded rather than HTML.. that can give trouble on non-standards-compliant browsers” – which won’t make much sense to the user without a fair amount of extra explaining about standards, formats and browsers – or he can *show* rather than tell the user how to avoid the problem another time. Naturally, as neither supporters nor users are usually patient enough for long explanations, the mimetic and more pragmatic approach is by far the most common – and this is also the usual way in which the rudimentary ‘recipe knowledge’ of the average user is developed.
With the seemingly vague, esoterically difficult language that defines its parts, and its near total opacity in its inner workings (being a grey box capable of seemingly magical feats), the computer appears to be inexplicable – much like the workings of the brain. It is this vagueness that often reinforces the ongoing anthropomorphisation of computers, already strong from popular notions of Artificial Intelligence stemming from the enthusiastic hopes for computing technology often offered by computer experts themselves in everything from pulp science fiction to academic journals.
In summary, the ordinary user's perception of the computer varies between not being consciously aware of it when using it as a ‘ready-to-hand’ tool (though this is also very much due to the fact that the working environment offered through the “desktop” metaphor decreases the abstract unfamiliarity of the machine, giving the user an intuitive feeling of greater control and less need to understand the esoteric, acronymic language of the computer) that simply is performing according to the expectations of the user's limited 'recipe knowledge', and perceiving it psychologically as a fussy “spirit” when it isn’t working as expected, needing to be appeased, in order to return it to a working shape or simply to make it do as the user want.

The hacker

So far, I've been setting up a sharp dichotomy between “average computer users” and “computer experts”, yet this distinction is crude. There are many computer users who don't anthropomorphize their computer, and there are many so-called experts, including myself, whose technical knowledge is quite limited, but who use their basic understanding of the inner workings of computers to give them a conceptual framework upon which to recognize and solve his problems. Separated from these users are the true computer experts, or hackers, as they like to call themselves. In the online “Jargon File”, a dictionary of hacker terms and expressions collectively compiled by computer enthusiasts, the term hacker is defined as “A person who enjoys exploring the details of programmable systems and how to stretch their capabilities, as opposed to most users, who prefer to learn only the minimum necessary.”[4] Similarly, the act of hacking in that sense is defined as the playfully clever exploration and experimenting with computers. Unlike ordinary users, who merely use the functions already programmed into the machine, hackers spend their time programming the computer to do their bidding, exploring new ways to use the computer.

Sherry Turkle likens hackers to virtuoso pianists or artists – they're all caught up with an intense need to master their medium, to understand and master the amazing complexity of the computer (Turkle 1984: 207-225). To hackers, the computer is not just a tool to be used in order achieve other means, such as writing and printing your essay, contacting your friends or calculating your taxes – it is a medium of expression in its own right. It is the process of winning and retaining this mastery over the computer that is the thrill of hacking.

When socialising, hackers often use jargon, much of it derived from computer acronyms and science fiction novels.[5] These are the esoteric mutterings that ordinary computer users often are exposed to when hackers or other experts fix their 'unready-to-hand' computers. But the programming languages that hackers spend most of their time using, cannot be spoken at all.

[4] Despite the common usage in the press of “hacker” to describe malicious meddlers who break into computer systems, most of the computer enthusiasts who label themselves hackers do not appreciate these connotations. Instead, they use the word “cracker”to describe such computer vandals (cf. the Jargon File entry on Hackers).

[5] Most parts of the computer have troublesome technical names such as RAM (Random Access Memory) or CPU (Central Processing Unit). See the Jargon File [http://www.catb.org/~esr/jargon/] for a dictionary of hacker's jargon and phrases or The Ultimate Computer Acronyms Archive for a list of computer acronyms [http://www.acronyms.ch/].

Computers are complex machines that only understand input of 0 and 1 (such as whether an electrical current is connected or not. This atomic unit of information is called a *bit*). This most basic of communication is called machine language by computer programmers. All the usable functions such as calculations, applications or protocols that we use everyday consist of this simple input. In the early days of computers, all programs were coded directly in this binary code which could take a long time, and was very unintuitive for the programmer. Programmers therefore developed various assembly languages which offered a list of commands more easily remembered than binary code, for instance to tell the computer to add two numbers, the machine language command might be *11001101*, while the assembly language command would be *add*. Still, assembly languages were very primitive with a limited set of instructions available, severely limiting what the programmer reasonably could make the computer do. To widen the possibilities of computer programming, hackers began designing high level programming languages that contain hundreds of different instructions, statements, and parameters to be fed to the computer. Using a high level programming language, the programmer can write a program in a form that is semi-readable by human being and which can much conceptually precise than assembly language. An example of a program written in a high level programming language, instructing the computer to put the words "Hello World!" on the computer screen, could look like this:

```
main()
{
    printf("Hello, World!\n");
}
```

Written in the "C" programming language, this "source code" cannot be run directly by the computer. First, it must be turned into machine language, or "object code" using another program called a compiler. Programming is thus not a direct interaction with the computer, but rather the writing of code that, when compiled, can be run by the computer.

C is just one of many high level programming languages available. There have been designed and programmed several thousand programming languages over the past 50 years, and new ones appear all the time. What a hacker can make a computer do is determined by the language with which he works, and though all programming languages are meant to be “Turing Complete” – i.e. capable of expressing all manner of computation – they use wildly different instructions to produce these computations, leaving the programs and code the programmer can design and produce very much dependent on the programming language. Designing a programming language is often a balance between making a language easy to learn (i.e. having simple, or natural-language-like commands) and expressive power (i.e. the precision and concision of the commands) (Wilson & Clark 1993: 318-319).[6]

Programming languages are unlike any natural language in that their communicative focus is the computer – a machine that only can do what it is programmed to do, nothing more, nothing less (Turkle 1984: 274). If the computer responds, it is either with a failure or a success message, depending solely on the foresight of the programmer. The programming language is thus centered on the program – i.e. a list of actions that the computer can take, depending on the input from the user. This requires a completely different syntax than that used in natural languages, as all the computer can understand is “if user chooses A, then do B (B consists of first doing C, then D, then E)”. This is syntactically extremely demanding, as most programming languages use only imperative or declarative commands, parentheses, brackets, indentations and punctuation to achieve its effects.[7] The computer needs

[6] There are three main groups of programming languages that use fundamentally different principles in communicating with the computer: These are called Imperative/object oriented, Functional and Logic languages – for more information see Wilson & Clark: “Comparative Programming Languages” (1993). A way to compare basic differences in syntax and indentation between programming languages is to look at the differences that appear in the above “Hello World!” example. See the webpage http://en.wikipedia.org/wiki/Hello_world_program for an extensive list of examples. For more on programming languages in general, see Paul Graham's essay, “Programming Languages Explained”, (Graham 2004).

[7] This is why one cannot “speak” a programming language. It is simply not meant to be spoken. Attempts have been made to make programming languages look and feel more like regular, spoken languages such as English, but since the computer rejects all linguistic ambiguity and interchangeability, these attempts have all failed. Some hackers enjoy writing poetry in programming languages, and one language better suited for this endeavour than most, is called Perl. Here is an (extreme) example of Perl poetry:

```
< >!*"#
^"`$$-
!*=@$_
%*<> ~#4
&[]../
|{!,,SYSTEM HALTED
```

Read aloud, that would be:

Waka waka bang star tick tick hash,
Caret quote back-tick dollar dollar dash,

mathematically precise instructions, or the program will fail, and even one mistake can jeopardize the entire output of a program.

The American computer scientist Gerald Weinberg has compared writing a computer program with praying – it is unidirectional, and it not until you have compiled the code that you will see whether it works or not (Weinberg 1971: 207).

Just as with the average computer user, the hacker normally perceives the computer as 'ready-to-hand' – but without the intermediate and limiting level of the graphical user interface. He types his commands directly and precisely for the computer to compile and run. It is in this way that the programmer actively and repeatedly seeks to test and confirm his ability with the computer, risking 'unreadiness-to-hand' – a rejection of his code – in hope to feel the thrill of mastering the computer.

Many computer scientists, among them Donald Knuth, author of the influential seven-volume “The Art of Computer Programming”, argues that well-crafted computer programs can be aesthetically pleasing, and that they can be joy for other programmers to read. According to Knuth, a beautiful program should be as mathematically precise and logically concise as possible, leaving less room for mistakes (or 'bugs', as they're called in hacker terminology), it should be programmed so that it will require as little processing power as possible to run, avoiding the inelegant solution of using massive amounts of computation to achieve limited results (Knuth 1992: 1-9). It is even possible to talk of distinctive *styles* of programming (Knuth 1992: 8-10).

Steven Levy describes how hackers compete among each other to “bum” instructions out of the code, making it as compact and streamlined as possible for the computer to run effortlessly (Levy 1994: 43-45). The way the computer responds to the program is the best indication of how well your programming skills are honed, of how sharp your mind is. The hacker uses his intellect to infuse life into the program/machine combination, and as the Russian computer scientist Andrei Ershov puts it, “this triumph of intellect is perhaps the strongest and most characteristic aspect of

Bang star equal at dollar under-score,
Percent star waka waka tilde number four,
Ampersand bracket bracket dot dot slash,
Pipes curly-bracket bang comma comma CRASH.

Which sounds more like a Dadaistic sound poem than any meaningful poetry. See http://www.perlmonks.org/index.pl?node=Perl%20Poetry for more examples.

programming" (Ershov 1972: 504). This is the essence of what Levy calls the "hacker ethic" – a collection of ethical notions that hackers seem to have in common: Boundless technical curiosity, desire to take apart and master new technology and share your knowledge,[8] belief that it is possible to create works of art and beauty on a computer (and that these can change your life for the better) and a meritocratic notion that hackers should be judged only on the quality of their hacking, rather than any other criteria such as age, academic degree or position (Levy 1984: 39-49). Hacker culture is thus centered on the computer – the final and impartial judge of your hacking and programming abilities.

Often, programs grow so complex, full of interrelated elements and pieces that the programmer will not have the sufficient time to explore all of it to master it fully. This will leave the programmer much in the same situation as an ordinary user, letting some amount of superstition of the psychological machine pervade the way he deals with his program. Again, like many ordinary users, the programmer is aware that this is wrong, yet he does it with the conviction that the computer is infallible, and any problem along the way can be solved if enough time is spent working on it (Vinge 2001: 19-20).

Because of these human factors in programming, the computer program becomes a reflection of the mind of the programmer, expressed in the code. This makes programming languages central in order to understand not only the hacker's interaction with the computer, but also his interaction in the playful cleverness and competitiveness of hacker culture through sharing of technology and beautiful code, and the collaboration on shared projects which is by far the norm in programming.[9] This means that when the hacker chooses his programming language (or even writes his own!), he is not just picking his tool of choice for his given task (some languages are good for small scripts – programs that quickly can solve a newly occurred problem, others are good for writing whole operating systems, works of immense complexity) but he is also picking his means of expressing himself, his aesthetic style

[8] It is this curiosity and joy of sharing that is the driving force behind the Open Source movement who advocates that a program's source code should be available for studying to all who might be interested. As it is today, most software companies do not offer the source code, but only the binary object code, which the computers can read, but is impossible to decipher for human beings.

[9] Weinberg argues that programming is indeed a social activity and cites that most programmers spent more than two thirds of their time working with other people rather than working alone (Weinberg 1971: 35).

– his mode of thought. As the renowned American computer scientist Alan Perlis puts it: “A [programming] language that doesn't affect the way you think about programming, is not worth knowing.” (Perlis 1982).
One of Sherry Turkle's informants described the act of programming thus:

> Some people don't program straight from their mind. They still have to consciously think about all the intermediate steps between a thought and its expression on a computer in a computer language. I have basically assimilated the process to the point that the computer is like an extension of my mind. Maybe of my body. I see it but I don't consciously think about using it. I think about the design, not implementation. Once I know in my mind exactly what I want to do, I can express it on a computer without much further conscious thought.
> I usually don't even hear in my mind the words that I am typing. I think and type ideas expressed in LISP [a programming language].

(Turkle 1984: 212).

This hacker has absorbed the possibilities and limits of his chosen programming language so well, that he knows exactly what he could do with it, so that he only has to focus on how to implement it while writing it. He is basically *fluent* in LISP. Steven Levy tells the story of how one hacker entertains two other hackers by coding assembly language tricks which they – with their shared mastery of the programming language – found to be “hilariously incisive jokes”, with every few lines of instruction leading to another punch line (Levy 1994: 136) – thus turning the programming language into a direct communicative means between people, thinking so well within the confines of the computer to understand the code as it is being written.
Another way of describing how a programming language structures the hacker’s perception of what can be done with the computer can be found in Gerald Weinberg’s “The Psychology of Computer Programming” (1971), where he begins his discussion of programming language design with the story of “Levine the Genius Tailor”:

> It seems that a man had gone to Levine to have a suit made cheaply, but when the suit was finished and he went to try it on, it didn't fit him at all. “Look,” he said, “the jacket is much too big in the back.”
> “No problem,” replied Levine, showing him how to hunch over his back to take up the slack in the jacket.
> “But then what about the right arm? It's three inches too long.”
> “No problem,” Levine repeated, demonstrating how, by leaning to one side and stretching out his right arm, the sleeve could be made to fit.
> “And what about these pants? The left leg is too short.”
> “No problem,” said Levine for the third time, and proceeded to teach him how to pull up his leg at the hip so that, though he limped badly, the suit appeared to fit.
> Having no more complaints, the man set off hobbling down the street, feeling slightly duped by Levine. Before he went two blocks, he was stopped by a stranger who said, “I beg your pardon, but is that a new suit you're wearing?”
> The man was a little pleased that someone had noticed his new suit, so he took no offense. “Yes it is,”

he replied. “Why do you ask?”
“Well, I'm in the market for a new suit. Who's your tailor?”
“It's Levine – right down the street.”
“Well, thanks very much,” said the stranger, hurrying off. “I do believe I'll go to Levine for *my* suit. Why, he must be a genius to fit a cripple like that.”
(Weinberg 1971:210-211).

Substitute the suit with a programming language, and Weinberg says that you have something close to the feeling a programmer experiences when he begins to learn a new programming language. Deciding to learn a new programming language is often a question of accepting that what the programmer wants to accomplish cannot be done in the language he already knows. In learning a new language, the programmer has to bend his mind from the familiar ways of his first language to the different limits and possibilities offered to him by the new one. At first he does not so much notice how well dressed he is, as how crippled he feels, yet the more programming languages he learns, the more he will be aware how one-sided his way of programming has been. Programming languages are attempts to make communication simpler between the computer and the programmer, seeking to give him greater expressive power with less mental investment, but so far it is still the programmer that must scrunch in his suit to fit the alien binary thinking of the computer (cf. Weinberg 1971: 211-214).
In short, hackers use programming languages to communicate with and master the computer, seeing it as an expressive medium in its own right in which they can display their technical mastery of complexity to their peers by making the computer do surprising feats and by creating aesthetically pleasing code. The hackers consider the computer a perfect machine, responding flawlessly to the often flawed input of ordinary people. Communicating with the computer is solely on the terms of the machine, forcing hackers to think in terms that the computer can understand. And as hackers become familiar with their tools and grow closer to the machine, they also come to perceive it psychologically, but unlike the ordinary computer user, the hacker perceive the machine as a perfect, unprejudiced subject that judges them solely on their intellectual capacity.

Language and our perception of computers

Cognitive science has long been inspired by the computer in attempting to produce a coherent way of understanding how the brain works. Not only is computer jargon full of 'mind jargon' inspired by the hopes of artificial intelligence and the continuing anthropomorphization of computers, but computer metaphors are also spreading to human mental terms. A teacher can say that he has his next lecture “hardwired”, meaning that he can deliver without thinking, a girl can say that she is in “debugging mode” when going to her psychotherapist, or a boy can say “his stack has overflowed”, meaning that he has lost his train of thought as other thoughts or ideas were crowding for attention (cf. Turkle 1984: 16-17). But as the British anthropologist Maurice Bloch has pointed out, recent cognitive studies has shown that our way of perceiving the world or “processing information” is markedly different from that of computers. Our everyday knowledge and thought is generally embodied and non-linguistic “scripts and schemata” – formed concepts that exist before language (Bloch 1998: 6). According to Bloch, this means that when our brain works to fit a concept to a word, a situation or an image, it does not work like a computer, processing one bit of information at a time, but rather recalls a highly flexible mental model or schema with which to compare the new situation (Ibid. 12). He argues that it takes a while for the human brain to build any such script or schemata – such as slowly learning to ride a bicycle or using a GUI. Much of this kind of knowledge cannot easily be communicated through language, only through interaction with the world – either by exploring on your own or by mimetically apprenticing to someone already an expert at that skill. In this way the “data” is registered, becoming habitually and easily available for use in our everyday negotiation of the world. This would fit with how the ordinary computer user generally perceives the computer according to two simple schemata: One of the computer working as 'ready-to-hand' with the user’s ‘recipe knowledge’, and one of the computer not working. Due to the opacity of the computer itself and the abstract acronymic, almost esoteric jargon used by experts to describe its parts and claim authority over the machine, the ordinary user will tend to identify the computer using a sort of “folk psychological” image of the computer as a fussy “spirit” – already half-necessitated to negotiate the shock of the computer's occasional 'unreadiness-to-hand.' In this way, the user comes to look upon the computer expert almost as a sort of “techno-shaman” who has mastered the esoteric, magical nature of the machine, solving problems through mystical rituals.

Hackers on the other hand, perceive the computer as more than a mere tool, but as an expressive medium to be mastered – a goal in its own right. Due to the inflexibility of the computer, the hackers are forced to think in certain patterns, set by their programming language of choice, in order to facilitate stable communication with the computer. In designing and constructing software, the hacker is creating abstract blueprints of enormous complexity with no physical referent except a few vague metaphors. The mental imagery used to explain the processes of the computer is a far cry from the dream of cyberspace that is still alive in the popular imagination, and it is solely up to the hacker’s own abstract imagination to make ‘stacks’, ‘ports’ and ‘links’ fit together into a working whole to be tested by the computer.
Programming as a skill seems to be a purely abstract interaction, with no direct worldly referent, and does as such not fit easily with Bloch's concept of cognitive schemata. Even though a hacker can adopt a programming language in such a way as a cognitive schemata, the greater his familiarity with this language becomes, the more it will limit his wider possibilities of mastering the computer. Programming, like other abstract, linguistic skills such as writing or scientific enquiry, cannot simply be learned by interacting with the world. In order to understand this, I suggest the American anthropological linguist Benjamin Lee Whorf's idea that some languages are better suited for communicating certain intangible ideas than other languages. His classic example is of the Hopi language of Southwestern USA, allow for the conceptualizing time as truly intangible, rather than the general tendency in English of dealing with time as a concrete, almost tangible substance to be measured and described spatially. Whorf's point is that concepts inherent in language can be so pervasive in this way, that we rarely notice how they limit our way of communication and thus maybe even our thinking and perception of the world (Whorf 1956: 134-159, Lucy & Wertsch 1987: 73-74). This seems very much the case with programmers who use intangible concepts not easily translatable – even between each other, much less to natural languages such as English. Like Levine’s suit, it is only when we try to learn something new that we realize how set much our perception and actions have been

defined by what we have used so far.

Interestingly enough, Sherry Turkle describes how children learning programming at an early age don't need to understand the mathematical principles behind programming to make the computer work. By experimenting with variables and commands, the children develop a mental model of what the computer is capable of, indirectly accepting the unequivocal rules it sets (Turkle 1984: 93-136). This may prove to be quite telling for how we learn our own natural languages and how our grammar and word association is defined by our cultural and social surroundings. So much of our experience, of what we perceive is intangible and cannot easily be expressed through language. We use language to communicate abstract and intangible things and ideas such as frustration, anger or love to each other, but we each have our own ideas of what that is, culturally bound to some degree. In this way, poets and programmers share the continuing struggle of finding words, using words to communicate this intangible meaning. Computer programs, like other abstract concepts, theories or ideas with so little physical referent that vague metaphors are our only way of describing them may indeed be structuring our perception of the world, especially our attempts to communicate that perception.

It is difficult (and will at least require much further study) to say whether it is the abstract idea, the linguistic definition and communication of that idea or the phenomenological using of language as a tool that actually shape our perception of the world, but as I hope to have shown in this crude “armchair” analysis, language certainly does influence our perception of the world to some degree.

Bibliography

- M.E.F. Bloch, “Language, Anthropology and Cognitive Science” in: M.E.F. Bloch, *How we think they think* (Oxford: Westview Press, 1998), pp. 3-21.
- A. Ershov, “Aesthetics and the Human Factor in Programming” in: *Communications of the ACM*, Vol. 15, no. 7 (July 1972), pp. 501-505.
- Paul Graham, *Hackers & Painters* (Sebastopol CA: O'Reilly, 2004).
- D. E. Knuth, “Computer Programming as an Art” in D.E. Knuth, *Literate Programming* (Stanford: Center for the Study of Language and Information, 1992), 1-16.
- S. Levy, *Hackers – Heroes of the Computer Revolution* (New York: Penguin Books, 1994).
- J.A. Lucy & J.V. Wertsch, “Vygotsky and Whorf: A comparative Analysis” in: Maya Hickmann (ed.): *Social and Functional Approaches to Language and Thought* (Orlando FL: Academic Press, 1987) pp. 67-85.
- A. Perlis, “Epigrams on Programming” in: *SIGPLAN Notices* 17(9) (September 1982). Also available at http://www.bio.cam.ac.uk/~mw263/Perlis_Epigrams.html
- S. Turkle, *The Second Self – Computers and the Human Spirit* (New York: Simon & Schuster, 1984).
- V. Vinge: “Introduction” in: James Frenkel (ed.): *True Names – and the Opening of the Cyberspace Frontier* (New York: Tor, 2001) pp. 15-23.
- G. M. Weinberg, *The Psychology of Computer Programming* (New York: Van Nostrand Reinhold, 1971).
- B.L. Whorf, *Language, Thought & Reality* (Cambridge MA: MIT Press, 1956).
- R. Willerslev, “Spirits as 'Ready-to-Hand'” in: *Anthropological Theory*, Vol. 4 (4) (2004), pp. 395-418.
- L.B. Wilson & R.G. Clark, *Comparative Programming Languages* (Wokingham: Addison-Wesley Publishers, 1993).