year's experience with simple ones. Theory set off from finite automata and regular languages and carefully worked towards the Turing machine as coronation of all efforts.
This is the approved way to build a solid pyramid of knowledge, and there is nothing wrong with it except that too many students tend to get frustrated. If one is fascinated by the Internet, robots, video games, or applications like Photoshop®, exercises with transistors and regular languages can be frustrating unless it is apparent what they are good for. Those who bring their study to a good end will appreciate—afterwards—that they were given a solid basis. But too many give up before they can ever see why they have to learn these stuffy subjects. And many of those who do *not* give up encounter the core courses of their curriculum as frustrating obstacles.
There is much to be said for bottom-up presentation of knowledge—as long as students know what all those subjects are good for and what it all will lead to. If they don't—why should they care about set theory, finite automata, a program that computes the factorial, and Boolean algebra in the first weeks? These subjects, boring to those not attracted by the beauty of mathematics, seem not even to have anything to do with each other—let alone with the fascination of the Internet. How can you ever be fascinated by the addressing modes of the instruction set of a certain processor if you have not experienced the usefulness of complex data types? How can you lay the link between hardware and software if the compilation and linking of your programming exercises is hidden by a didactic programming environment, until you see the first compiler of your academic life in an advanced course on compiler construction?
The authors' hypothesis was that a small addition to the curriculum might prevent many problems: a short, compact introductory course at the very beginning *that explores the pyramid of knowledge upside-down*.
This approach is not uncommon in other academic disciplines. For centuries students of medicine have started by dissecting a corpse, outside in. From the complex but well-known to the small and unknown. Before they understand about physiology, orthopedy, or which specialisation ever, they get the chance to learn, in a relatively short time, how the complete human body is organised, from what you see at the surface down to the level of tissue. They learn to get oriented in the main subject of the discipline they want to study.
At many universities the classical study of natural science starts with a big and impressive "circus", where as many as possible of the fascinating phenomena of physics are actually demonstrated in the lecture hall. Students see the richness and fascination as they will hardly ever see it later.
The question is: can something similar be done for informatics? An introductory course that demonstrates the fascinating phenomena that are studied and (other than in the case of medicine) created by informatics—in all their richness, starting from the complex and well-known surface, gradually going deeper to the small and simple, but unknown.
To get some evidence, a one-week course "Introduction to Informatics" was developed and given. A more speaking title would have been "Anatomy of computer systems", but the Department of Informatics considered that too revolutionary.
The full course, as it was actually given first in 1998, had to meet extra goals not covered by this paper. It had to make another course superfluous, which tried to prepare students for the pitfalls of academic learning and to teach them practical skills in using the department's computer systems. These aspects are not covered by the present paper, which attempts to explain the philosophy and architecture of the "light" version of the course and to show what has been achieved during the first time it was given: how did it go, what have the students learned, did they like it?
After a brief statement of the goals and methods we present a chronicle of what actually happened from day to day.
Although we hope to inspire and encourage lecturers to follow a similar concept in teaching, this report is not meant to be a didactical text on how to plan and give such courses. In a later publication we shall explain in detail how we planned and prepared it.

Goals of the course

The full course had to meet four goals in the rather short time of one week. Only the first two of these are covered by the present paper.

1. Orientation in the world of phenomena

Informaticians deal with very different concepts like: program, network, hardware, protocol, algorithm, compiler, byte, specification, test,… Informatics studies a rich and very complex world of phenomena, and to beginning students this world can be frightening or frustration due to a lack of orientation.

The "Taxonomy for Computing Science" (WUPPER and MEIJER, 1997) has successfully been used as a means of orientation in teaching in Nijmegen for a number of years. It was a corner stone of the course described here. Nevertheless, taxonomy has not been taught explicitly in this course. Like in introductory courses to biology, geography, or any other science, phenomenology, morphology, and topography have to come first. We cannot start to order and classify phenomena before we have learned to distinguish and to describe them.

So the first goal was the demonstration of the richness of phenomena and their underling relation. Take protocols, for example: students should lean to know where to look for them and identify them when they come in disguise.

2. Experience of informatics as a science

Informatics is more than an enhanced programming hobby—it is a genuine science (or at least, hopefully, developing to become one). Informaticians are not just hacking along until they have written an impressive program—they are researchers.

So the second goal was an exposition of some exemplary research questions, both solved and unsolved ones, as well as the rôle of mathematics, logic, languages, formal methods in solving them. These should be chosen in such a way that pointers to all courses in the undergraduate curriculum can be given.

3. Forms of academic learning

Teaching and learning at a university is different from what first year's student, who just have left school, are used to. Students are faced with new forms of teaching and can easily suffer from a heavy culture clash.

4. Practical skills

In order to follow the regular courses efficiently, students need to be able to use the Internet (e-mail, news, www, irc), to compile and run programs, and, in general, to find their way through several different operating systems. Many beginning students already have gained a lot of experience with their private computer system at home, while others do not have any experience at all. Those who already have experience are not always able to efficiently use it when confronted with another type of system.

So the fourth and last goal was to achieve the necessary practical skills like, for example, to communicate with a lecturer efficiently and professionally by e-mail or to layout and print a readable report.

Method

Most courses in mathematics, theoretical informatics, programming, or computer architecture follow the opposite direction, slowly and carefully building up something powerful and complex from the most basic elements. Our course followed the opposite direction. The main idea can be described in one sentence:

Let the students dissect the object of the course, working from the surface inwards, and make them observe and understand the phenomena they come along.

At the surface of contemporary computer systems one sees screens, keyboards, loudspeakers, printers, etc. connected in a world wide network, performing useful functions. And that is from where the course worked down to the level of electronic switching elements, viz., logical gates.

A prerequisite for success of this approach is that the teachers help the students not to get lost in detail questions. Students have to learn, as fast as possible, that the phenomena at one level can be understood "conditionally", i.e. by temporarily assuming some hypotheses about the next lower level, which will not be investigated until later. (It is not difficult to understand how the internet technically works, *provided* we assume, for the time being, that computers can be made to do whatever is required and that they can exchange information with other computers. At the present level we can then concentrate on the basic ideas behind the Internet Protocol.)

Therefore the teachers need to have a good understanding of the layered nature of the object of the course (the "Chinese Box Principle", cf. WUPPER and MEIJER, 1997). They have to be continuously aware that questions that belong to a lower layer should be made explicit with sufficient precision *but not answered* until the layer to which they belong will be investigated. ("Yes, but how does such a data packet *look exactly* on a cable?" – "Never mind; we will see that the day after tomorrow. But keep in mind that it contains the address of the receiver, which is unique all over the world.")

Another prerequisite is that the whole course is driven by questions rather than answers.

Technodrama

A way to understand the essence of a sense-less but efficient mechanism is to have to perform the function of one of its mechanical parts while at the same time being a human observer of the whole. To help a group of students understand the working of a machine we can let the group become to *be* that machine and at the same time let them find out its principles. In the end of the learning process everybody must perform a well-defined mechanical task governed by clear rules.

This approach resembles the 'psychodrama' [Moreno 87] well known to psychologists. Participants of psychodrama experience how everybody can easily become a quasi-mechanical part of a pathological constellation in a family, at work, or wherever and learn to understand what one can do to escape that mechanism. In our approach, which we will call "technodrama", the participants learn to understand how it is possible that a senseless mechanism can perform tasks so complex that intuition does not help, while at the same time they experience the difference between human creativity and strictly mechanical action.

The author accidentally invented technodrama when he once had only one hour to make a class of ambitious students of economy understand the basic principles of computers. The students were divided into memory cells, a program counter, someone who could fetch information from a given memory address, an instruction decoder, an adder, etc. Soon, mechanical computing had lost its secrets. An unforeseen side-effect was that these students of economy became particularly motivated to discuss, in another course, the inhuman aspects of Taylorism.

The idea of technodrama is so simple that the author expects it to be, under whatever name, a well-known didactic technique. Nevertheless he has never come across it in his own career as student and teacher of informatics.

Structure of the daily lessons

All five the course days were planned to have identical structure, except that the last afternoon ended with drinks in the faculty garden. The mornings were spent plenary in a lecture hall. In the afternoons the students exercised in the lab.

1. Brainstorm

The morning started with a brainstorm around the theme of the day: What *is* it essentially? What are the parts from which it is composed? How can the essence be captured in some simple statements?
From Tuesday onwards, every brainstorm started from where the one on the previous day had stopped, taking into account what had been learned during the previous day.

2. Phenomenology

The second section consisted of a demonstration of a wide spectrum of phenomena. The guiding questions were: "what is essential?" and: "what is the state of the art?" Various examples of hardware were physically exhibited in the lecture hall, while the software demonstrations were done by means of a Macintosh PowerBook (laptop) connected to the Internet and to a beamer projector.

3. Principles

The third and most important section, again in discussion form, was dedicated to analysis of the phenomena seen. How is it possible that this works? What is necessary to make it work? How can we learn to structure what we see? Can we understand it in a simple way, without getting lost in detail?
Not only principles from informatics (protocols, programming languages, data structures, etc.) were addressed. Attention was also explicitly paid to more general tools of academic reasoning, like abstraction, structuring, Occam's razor, languages, recursive definitions, etc. whenever concrete instances of these passed along.

4. Exemplary research questions

The fourth section consisted of the presentation and discussion of one or two exemplary research questions. Why is it far from trivial to understand or make this? What are the pitfalls? Which achievements of mathematics and science are necessary to solve this?

The five days

This is a chronicle of what happened in the lecture hall from day to day. The reader will notice that the line of discussion was influenced strongly by the students' questions and observations. With some careful stimulation they could detect many essential questions by themselves.
The phenomena demonstrated and discussed had been carefully selected and prepared in advance, of course. In this chronicle, they are marked by "→" in order to give an impression of the planning. Of course, the selection depended on what was available. The teachers of this course should feel free to adapt the list to the local possibilities—as long as the spectrum remains broad.
The principles of the tool-box of academic reasoning that came along are marked by "•".

1. Monday. Networks

The lecturers introduce themselves to the new students. One of them does this via an → amateur video conferencing connection with CU-SeeMe and a QuickCam, demonstrating some of the possibilities but also limitations of this technology.

Thereafter, goals and the structure of the course are explained briefly. Emphasis is put on the importance of discussion, questions, and critical remarks. Scientists are people who love to formulate *difficult questions* and to challenge each other, and we are now starting together to exercise this. Do not hesitate to ask any question whatsoever. You may find that you already know much of what is demonstrated here. But are you sure that you can explain it to your aunt? Let us try to understand what informatics is al about, and, above all, let us try to separate the essential from incidentals.
After that—brief—introduction everybody can get a cup of coffee, and the proper programme will start.

1.1. Brainstorm

Today we will investigate computer networks. What *is* a computer network? To stimulate the brainstorm, five different → networks are—at least to some extent—present in the lecture hall: (1) a Macintosh → PowerBook, connected to the → Internet, to a → camera and to a → beame; at closer notice the PowerBook turns out to be a network by itself, containing various → PC-cards and more. (2) a → radio clock receiving time signals via long wave from Frankfurt. (3) a → banking card plus a small → chip-card reader to be worn on a key ring. (4) the campus → telephone network. (5) the → GSM-telephone network.

What are we studying?
What, essentially, is a computer network? Obviously the physics of information transmission can vary: it can be done via all kinds of cables or via radio. There even are → packet radio satellites that physically carry computer data around half the Earth, to let them "drop" on the place of destination.
The common aspect is that a variety of computers exchange information along suitable channels. The specific nature of the connections tends to become less and less important: Internet can come via tv cable, → telephone, → Ethernet, → GSM or whatever.

What are its components?
What, then, are the "computers" that form the *nodes* of a network? They can come in extremely different shapes and sizes, but obviously they al have something in common. What is that? It has to do with information processing and with communication; most of them are programmable in some way or other; in any case they are objects designed for a purpose. Whether they have a keyboard and a screen is less essential. Even a modem turns out to be a complete computer at closer look.

1.2. Phenomenology

A number of Internet-applications are demonstrated and discussed. It turns out that, although being impressive, most of them are nothing new.
Under certain reasonable assumptions most of them can be understood easily. *Provided* that the Internet is a network that can transport all kinds of information between any two computers and *provided* that a computer can be programmed to behave as almost any other technical device, → Internet-phone is just telephone, → news is essentially nothing but a clever pin-board, → irc is a textual version of the chat boxes known from telephone, → e-mail is mail. Tell me what kind of communication you need, and it can be done via Internet.
One Internet-application, however, brings something completely new into the world: the → world-wide web with its clickable hyperlinks and its anarchic structure. Nevertheless, it is based on a couple of principles which are not difficult to understand.

1.3. Principles

For a clear discussion of the essence of the Internet it is important to stay at the highest level. If we look at bits on cables and in computer programs the great lines get out of sight.
• *Abstraction* is the first example from the toolbox of academic reasoning.
So let us assume that computers somehow or other *can* be programmed to do what we want them to do and that messages can be exchanged between two adjacent nodes in a network. What, then, is necessary to make an Internet?
The following basic ideas are explained and discussed.

IP-addresses
and their hierarchic structure which allows addressing of the whole world without one central office that gives out all addresses. Compare: bar codes, ISBN, telephone numbers.

TCP/IP
as a transparent protocol for all kinds of information. Data packets with address and type.

Name servers and routers
to connect the whole world without having a central office.

Uniform resource locators
as a means to address an arbitrary resource within an arbitrary computer.

Internet-applications
to make *meaningful* use of that general, transparent network. To communicate via the net, the partners need (1) a computer with Internet access and (2) suitable application programs to allow the form of communication they want.

1.4. Exemplary research questions

In the old days, a system administrator had to provide each computer in a network with a table describing the network structure and the nature of all nodes. Whenever the network was changed, all tables had to be changed. More elegant is a network in which all nodes configure themselves by finding out what their neighbours are. After some reasoning it becomes clear that when the nodes of such a network are switched on, they will have to elect a leader who can take some decisions for the whole network. How can all nodes be provided with identical programs in such a way that they can be guaranteed to decide on a leader? The discussion of this question brings a number of essential concepts and problems to consciousness: specification vs. design; verification; the importance of protocols; algorithms vs. concrete programs, etc.
The participants experience that it is a long way from the general idea ("just determine the computer with the highest serial number and make it the leader") to a protocol which actually achieves it, and that it might be even more difficult to actually prove that such a protocol is correct.

2. Tuesday. Application programs

Yesterday we attempted to understand networks. Later in the curriculum, a whole course will be dedicated to computer networks. Let us now focus on the nodes in a network: computers.

2.1. Brainstorm

What are we studying?
This day starts from a revised version of the first tentative characterisation of the concept 'computer': man-made, programmable, suitable for information processing and communication along a fixed number of ports, containing a memory.
From this definition the lecturer can easily prove that an → empty shoe-box is a computer: zero I/O-channels is a special case of a fixed number of communication ports. Obviously, the shoe-box is programmable for all possible functions that map no input to no output.
Clearly, an answer like "yes" or "no" to the question whether a shoe-box is a computer is quite worthless. A meaningful answer requires • *justification*, the second example from the toolbox of academic reasoning. "For me, this box is a computer *because*…"
To stimulate the brainstorm, the configuration around a typical → personal computer is investigated, in this case a PowerBook with peripherals as → printer, → Zip-drive, → ISDN-adapter, etc. It turns out to be completely irrelevant whether a → modem is built into the main box or standing next to it: it will in any case be a small processor connected by a cable. Our personal configuration with printer and external disk is nothing but a personal network, and inside the "main frame" we find more network, consisting of a → video-card, various PC-cards, possibly some co-processors, one or more main processor chips, internal modems and possibly more.
But it is even more confusing. "Intelligent printers" contain a processor that controls them, but printers can also be controlled by drivers running on the main processor in parallel with

application programs. And the newest processors are so fast that a separate modem is no longer required: along with whatever they have to do they can easily also convert digital information to sound and vice versa. Is there a meaningful distinction between one processor and a network? Or between hard- and software?

What are its components?
A computer consists of hardware and software. The central question of this day shall be: provided we have programmable hardware—how can the software be organised. Like yesterday, it is important to consider this at the highest level. How, essentially, is an application program organised and why can several of them be executed "in parallel" on one computer?

2.2. Phenomenology

Several typical state-of-the-art application programs are demonstrated: → Adobe Photoshop®, → FileMaker Pro®, → Apple QuickTime Virtual Reality®, → Now UpToDate®. The participants already know a lot of *Internet* applications from the previous day and they are to see more applications in the afternoon (text processor, spreadsheet, on-line manuals, etc.). The last group of examples shows that a sufficiently strong computer can be used to simulate virtually any other electronic device, for example an audio CD-player or a tv set.
How can all this be made to work?
In the case of the database system one can get a glimpse: when a new student's data are entered, a button "take photo" appears. This button is connected to a 'script' written in the FileMaker Pro scripting language. Investigation of this script teaches us that it 'calls' another script, written in Apple scripting language. And that script orders a certain application to take a photo and put in on the clipboard, whereupon the first script pasts it into the database.
The long and tedious way of information from input to output and through networks is analysed. How do pictures get into PhotoShop? How does a piece of e-mail text get from A to B? How does digital information exactly get through an old-fashioned analogue modem and then through a modern digital telephone exchange in such a way that the computer at the other end still recognises it?

2.3. Principles

Representation and conversion
Clearly an important part of the problem is conversion between many different *representations* of the same information—representations which often do not resemble each other in the least, or which are not identical but resemble each other so much that errors are easily made.

System structure
Whereas a computer network has a clearly visible topological structure, a (compiled) application program is an amorphous mass of bits. The objective of the discussion of this day is the power of
• *structuring*,—the third example from the toolbox of academic reasoning. When we know suitable principles, we can impose structure upon that amorphous mass of bits in order to understand it.

Communicating processes
At this point of the discussion the students are asked to assume, for the time being, that by some clever tricks to be investigated later a computer can be made to execute several independent programs in parallel. That if programs are given that could run on a number of computers communicating in a network, these same programs can be made to run, quasi in parallel, on one sufficiently fast computer, provided that some extra software for multiprogramming is added. Under this assumption, the confusingly ambiguous distinction between hardware and software becomes irrelevant. The treatment of physical hardware ("how can we make programmable machines?") can therefore safely be postponed until Friday.

Networks of functions and modules
So the question of this day has become: how can software for complex application programs be understood as a network of communicating processes?
Each piece of software can be understood as a box with a fixed number of communication channels for which well-defined representation conventions hold. Many such boxes are connected to each

other. If we look into a box, we will usually find that it is composed from a number of small boxes. Some of these boxes are mainly dealing with conversion of information between different representations. Others are dealing with protocols at both ends of a communication channel. Others do the "real work", like changing the contrast of a picture in PhotoShop, computing 2D-co-ordinates from a 3-dimensional virtual reality object, searching a database with a difficult query or checking the spelling of a text. The better defined and the simpler the tasks of the different boxes are, the better we can convince ourselves that the whole of them does what it should do. And if we want to understand a given program, we should know what to look for and consciously search for the necessary boxes.

Program structure as mirror image of network structure
A good way to understand the structure of a complex software system is to search for a mirror image of the hardware network that the system has to deal with.
A video-conferencing program has to deal with: an Internet connection, a local and a remote camera, a local and a remote keyboard, ditto microphones and loudspeakers, as well as windows for al of them on the screen. There is a great chance that each of these is mirrored by an individual software box and that the communication channels between these boxes mirror the physical channels between the hardware components. If it is not the case, the system will be badly written and error-prone.
As a second example, consider a big banking system. Every client (i.e. a person holding a bank account!), every money machine, every terminal in the bank, multiplexer or whatever will have its mirror image in the software. If we know what to look for, big software systems become less obscure.

Configuration management
As an exercise, the probable structure of the demonstrated FileMaker Pro database system is analysed. This leads to a new insight: the students detect that it is useful to distinguish between modules for "normal" usage (updating the database, queries) and reconfiguration (like adding extra fields or new views).

2.4. Exemplary research questions
Some research areas were mentioned briefly (relational databases, compression of pictures and sound, cryptography), but there was no time left to have a closer look at research questions.

3. Wednesday. Operating systems
Yesterday we attempted to understand the structure of complex software systems. Most of the curriculum is dedicated to the development and verification of software systems. Let us now focus on the questions that were left open yesterday and the day before: How is it possible that one single computer can deal with all those complex tasks?

3.1. Brainstorm

What are we studying?
Every well-written piece of software is a network of functions and modules. We have seen what powerful applications can be composed from such modules. We have got an idea that anything can be performed by a computer program *as long as we know exactly how it must be done.*
Aside: the lecturer had a dream in which different incidents of the previous day occurred in a surprising connection. The human brain obviously is very good at *associations* of formerly unrelated concepts, and we do not know how this works. Computer programs, on the contrary, are usually based on a divide-and-conquer approach.

What are its components?
How can one single piece of hardware execute several programs in parallel? What happens behind the screens if the "user" of a computer starts a program by simply clicking on an icon or entering a command?
The long and tedious way from a program as "executable" in a file until that program actually is being executed is analysed. Obviously, programs need to be loaded and started by other programs.

But if a program is already loaded—what does it mean to 'start' it? And what exactly happens if a program has finished its task. It 'stops'—but what does that mean?
For better understanding we nee some idea of how a CPU works.

3.2. Phenomenology

Hardware
It would have been good to have a → simulator for a very simple machine language; but the only simulator available at the time of the course is a SPARC-simulator. That architecture is clever, but too complex to demonstrate the essence of the Von Neumann architecture. Moreover, the laser-beam has too few pixels to project all the necessary windows on the wall. Therefore, a → very simple machine architecture is presented on the blackboard. It turns out that most students, even those who own a computer, did not have clear ideas of what a → machine instruction and a → program counter is.

System software
Then, the → system folder of the MacintoshOS is investigated. The → loader and other important modules turn out to be encapsulated in one monolithic program called → 'System', but many of the modules we should expect can be found in the extensions folder, among them → device drivers and → protocols.

Graphical user interfaces
The → 'Finder' is a → graphical user interface. It allows to *use* the system, i.e. to move files around and to start programs; but it also serves to *configure* the system by adding or removing system components. In fact, the system can be seen as a kind of database, and there is maybe no fundamental difference between the Finder and the graphical user interface of FileMaker Pro. Windows 98 teaches us that there is no need to make a difference between the desktop and a web-page—maybe within some years there will be just one generally accepted graphical user interface for web pages, operating systems, databases and anything else. And probably this will allow different 'styles' as a matter of taste, like it is already the case with UNIX' → X-windows.

3.3. Principles

User interface
Modern operating systems provide graphical user interfaces that do not differ essentially from the user interfaces of powerful applications. These interfaces (1) give access to information, (2) allow to command certain services, (3) allow to change information, and (2) allow to change the configuration of the system altogether. Usually only privileged users may do the latter.
Operating systems also have a *programming interface*: here not the user but programs running on the machine can ask for information and issue commands.

Memory hierarchy
We want computer memory to be large, fast, and cheap. Regrettably, the price per bit increases exponentially with access speed. Many computer systems therefore support a hierarchy of different memory technologies, from large but slow up to fast but expensive. We can distinguish at least: → cpu registers, → cache, → main memory, → hard disks, → zip disks, → streamer tapes. What are their respective characteristics (speed, price per bit, access method, addressing, etc.)?

Loader
The philosophically most spectacular component of an operating system is the → loader. It reads the → representation of a machine program from a → file on → data storage and writes an → executable program into → main memory. The loader itself is a program. Programs can be—and usually are—created by means of other programs. Where did it all start?

Data structure
Data structures can be found in computer systems at various levels of the memory hierarchy, but we can also recognize them outside computer systems. Often we find → nested data structures (→ records, → folders inside folders). The strongly hierarchic structure enforced by nesting can be by-

passed by → pointers or → aliases. Structured data have to be mapped to linear memory in some way, which is far from trivial.

Multiprogramming
On one computer a number of programs can be executed *quasi in parallel*, as if each of them had its own computer. A long time is spent on a technodrama session modelling multiprogramming.

3.4. Exemplary research questions

We have seen that there is still a lot of work do be done around operating systems which is far from trivial. Standardisation of interfaces and modularization are rather straightforward examples. During the technodrama session we have got an impression of the problems of → scheduling; we can understand that good solutions will involve a thorough mathematical analysis.
→ Garbage collection is an example that can easily be visualised. The hard disk of the laptop is defragmented by means of a → utility that continuously shows a map of the disk content on the screen. While this process is going on, the students are challenged to think about an algorithm that is efficient and at the same time so robust that at no time a system crash will lead to an inconsistent disk structure. What *are* the problems to be aware of and how could they be approached?

4. Thursday. Languages

Yesterday we have seen how a computer together with an operating system can allow programs to be executed. Tomorrow we shall see how the hardware of a computer can be made. Today, the question how programs come into existence will be investigated.
• *Language*—not so much the existing programming languages but the professional design and use of suitable languages for various goals— will turn out to be very important means from the toolbox of academic reasoning. The value of professional use of languages cannot be overestimated. Affinity with language phenomena is highly desirable for informaticians.

4.1. Brainstorm

What are we studying?
We have seen and used a lot of programs. But what *is* a program? Programs come in different appearances, and it can be confusing no to distinguish between them.
We have encountered → binary machine programs in working memory; these are long sequences of bits stored in machine words. If the program counter points to the wrong place, whatever stands there will be interpreted as a program. A whole course, "computer architecture" will be devoted to machine programs and their processing, later in the curriculum.
We have also seen → images of executable programs stored in files on hard disk, where a → loader can find them.
Today we shall see many examples of → programs written in some or other → programming language. These have to be *compiled* to make them executable.
When we tried to design a leader election protocol, we were actually working on an → *algorithm* without thinking of a specific language.
There are many different representations of the same program. And there are many different languages. What is a "programming language"? How are programs "made"? The long and tedious way from the idea for an application to the 'binary executable' on the hard disk is investigated and sketched on the blackboard.

What are its components?
Machine programs consist of bits, but it is better to regard them as consisting of instructions which themselves are stored in binary machine words, i.e. to impose structure on a higher level. Indeed, we have identified much more structure at even higher levels: a computer in the Internet contains applications and an operating system, which all are composed from modules, which are composed from smaller modules, etc., until we reach the instruction level.
In the representation of all these programs in higher-level programming languages this structure can be clearly visible.

Let us investigate the phenomenon of *language.* A language obviously contains some → primitive constructs but also means for → composition. Let us look at some different languages to get more insight.

4.2. Phenomenology

In modern computer applications, programs written in a variety of languages "call" each other. We have seen a → database system for the administration of our own course. It can take and store photos of the students. Let us look inside. When we press the "take photo" button, a → "script" in FileMaker Pro → scripting language is executed. This script calls an → AppleScript written in a completely different language, which sends commands to the → camera program.
It can be enlightening to compare programming languages with some seemingly vary different languages: → Midi, → traffic signs, and → the way a dinner table is laid. Do these deserve to be called 'language' at all? Yes: they are subject to strict → syntactical rules and have a well-defined → semantics. A syntactically correct arrangement of plates, glasses, spoons, knives and forks on a dinner table tells us exactly how many courses of what kind we have to expect in which sequence.
Not all programming languages consist of *text*: → LavView® is an example for a *graphical language* visualising the principle of → data flow.

4.3. Principles

Syntax and semantics
To use a language we must know its *syntax*, i.e. the rules how correct expressions may be formed, and its *semantics*, which determines the meaning of syntactically correct expressions. In natural language linguistics, syntax and semantics are far from being understood completely. This ought to be different with languages designed to be processed by computers: we should refuse to use them unless syntax and semantics have been clearly and unambiguously been defined.

Types of languages
Artificial languages are designed for different purposes. Programming languages, for example, are needed to tell a computer *how* it has to work. We have seen enough examples to believe that it can be very difficult to conclude from a program *what* it will do. In a course on theoretical computer science we will see later that it can even be principally impossible. Therefore, → *specification languages* are needed to state clearly *what* has to be done (for example: "Elect a leader and make sure that everyone knows who it is").

Language processing
Texts written in a suitable language can be → *interpreted* by humans or by machines. They can also be → *translated* to another language. What does that mean?

Bootstrap
Imagine you have invented the first really good programming languages and you have written a compiler that translates it to the machine language of your own computer. You want people all over the world to be able to use them on all kinds of computers, but you cannot write all the necessary compilers. What can you do to allow them to obtain compilers with a minimum effort for them and for you? Is it a ridiculous idea to write a compiler for you language *in that language itself*? Is a divide-and-conquer approach feasible, that allows the bulk of the compiler to be written once and for all machines, while only a small *code generator* has to be newly developed per target machine? Can most of the work be done on the target machine, and can as much as possible be done in that new language?
We try to understand the problem and some possible solutions. They involve one or more → *intermediate languages* and lots of compilers between these, most of them obtained by compiling each other. It is as good as impossible to keep track of such a bootstrap process without notational support. One needs a (simple) • *formalism* to express clearly and unambiguously which compilers can be obtained by which compilations and to calculate what may be missing.

4.4. Exemplary research questions

There are basically three research questions about languages: Which languages do we need? What are the best machines to execute them? And: How can languages be translated to each other?
Which languages *do* we need? In the early days of computers, languages more or less developed accidentally. These were then successively "improved" or "extended", which striving usually introduced more problems than it solved. Later, much research has been done on → high-level programming languages, which were hoped to solve the "software crisis". But can we hope to find The Optimal Computer Language like many people hope to find The Abominable Snowman? There is evidence that "powerful" languages, designed in order make programming easy, can make • *verification* of programs very difficult if not impossible. Even worse: such languages can introduce extra faults in complex systems. There is still a lot of work to be done around language design, but now we understand that rather than hoping to find The Ideal Language, ICT professions ought to be trained to choose or design simple but efficient languages tailored to the problem they want to solve.
What *are* the best programmable machines? Is the question of the research area of computer architecture. We will come back to that tomorrow.
One of the best-understood area of computer science is → compiler construction, focusing on the question how languages can be translated. If nowadays we want to write a compiler that translates a given language into another one, we can build on an enormous amount of useful theory, methods, compiler construction languages, and tools.

5. Friday. Hardware

5.1. Brainstorm

What are we studying?
We have learned a lot about software during the last days; now it is time to have a closer look at the hardware. What *is* a programmable machine? Is it necessarily electronic? No, the first computers were mechanical, later ones worked with relays. The specific hardware technology does not matter so much; we better concentrate on the *function* of the components of a processor.

What are its components?
From the technodrama session on Wednesday we have a first idea how a computer executes a program. Also, we know about working memory, the program counter, and other registers. Can we draw a diagram of the overall structure of computer hardware? A somewhat confused discussion tends to go too deep into the details of specific architectures the students had heard about. We must again do our best to stay at the right level. In the end we can agree that any programmable general purpose machine must have at least: a memory to store the program, a memory for intermediate results, some means of communication, and a *central processing unit*.

5.2. Phenomenology

An existing machine architecture
The hardware of a → cpu of a modern computer is so small that we do not see anything when we look at it with our bare eyes. If we look at it with an electron microscope, it will turn out to be so complex that we do not recognize anything. If we want to see anything meaningful at all, we must use a → simulator that visualises the change of bits in memory and registers of a machine. We have a → SPARC® simulator available, from which we can get an impression how complex contemporary machine architecture is and what happens when little program fragments are executed. But we still do not know how a cpu can possibly be *made*.

Designing our own cpu
To get more insight we can try to build our own machine. Technodrama is a convenient way to design a complex machine top-down. When we have agreed about a system component at a high level, a participant can perform the function of it. We need not design its internal structure for the time being. The other participants have to watch the behaviour of the human component carefully,

to ensure that it behaves absolutely mechanical, i.e. that it only does what its specification orders it to do.
The lecturer invents a very simple *ad hoc* machine language and writes it on the blackboard. A student can be found who writes a short program without telling us what it is to do. We decide on the roles → memory, → program counter, → memory access, → instruction decoding, → arithmetic, → comparison. After some confusion we see that these roles are not uniquely determined. Several different solutions can be imagined. It is essential that we make choices and then stick to them.
The programmer puts his program into memory; the lecturer makes the program counter point to the first location and starts the machine.
It takes some time until the audience has decided about a consistent system of roles and until the players have learned to behave mechanically conforming to their roles without being creative. In this process some roles are split into two, others turn out to be superfluous. Some participants get fascinated in optimisation and, with only slight stimulation, invent some forms of → parallelism and → pipelining.
In the end it is clear that a set of machine components could be specified that together form a programmable computer. How these can be built in hardware is still open.

5.3. Principles

Instead of designing hardware for all the machine components we have identified, we open the toolbox of academic reasoning again and attempt • *simplification* in order to *reduce the problem to its most simple form*. What is essential? How can a divide-and-conquer approach help? If we find out how to build a very simple but universal computer, we can always improve the design later.

Input / output
A cpu is a box with a number of → input and output ports. Inside the box instruction after instruction is executed, depending on the program counter. At the input ports values arrive from time to time; at the output ports values are produced from time to time. Remembering our investigation of networks on Monday morning, we know that some protocol must be involved to distinguish valid information on the cables from irrelevant noise when there is nothing to be communicated.
Also, we must know how the values are to be represented electrically.
Can we do al we need with a number of wires at which, at any moment, only one of the two Boolean values is represented? Yes. With n of such wires how many different values can be coded at any moment? The answer is 2^n. And one extra wire is needed for the most elementary protocol that allows to distinguish valid information from silence.

The instruction cycle
A cpu is a box with a sufficient number of binary input and output channels. It contains memory in which programs and intermediate results are stored. For simplicity we can realise the PC in a location of that same memory. The effect of one execution cycle then depends entirely on the actual values on the input ports and the actual information in the memory. And that effect consists in the updating of the memory and the production of values at all output ports. Mathematically it can be captured in one single function f:
(new memory contents, output) := f(old memory contents, input)
A computer is a box computing the value of that f again and again. In order to build a computer we have to solve three problems: how to realise memory, how to find the right f, and how to realise such an f in hardware.
Finding the right f is difficult. A whole course, later in the curriculum, will be devoted to it. For the time being, we will leave it for what it is.

Memory
We are about to design the simples possible cpu hardware. What do we need to realise memory? What is the *purpose* of memory? It is to preserve bits between the end of one instruction cycle and the beginning of the next one. We already hav input and output channels; could the effect of memory be achieved by means of them? Yes; it may look silly, but n memory bits can be realised by n extra output ports that are "fed back" by individual cables to n extra input ports.

A computer is essentially nothing but a box that, for a fixed, carefully chosen *f* again and again computes output := *f*(input), with most of the output ports are fed back to corresponding input ports to obtain memory.

Boolean functions
We still do not know how we can make such a box. By the way: how many different functions *f* do exist, mathematically, that map *m* input bits to *n* output bits? Some time is spent on an intermezzo on • *combinatorics*. It is always good to know how large the space is in which we search for a solution.
All we have to find out is how to realise in hardware any function from *m* Boolean values to *one* Boolean value. The overall problem can then be solved by a combination of *n* such functions.
Let us assume we have three sorts of electronic elements with one or two inputs and one output. One sort 'computes' the negation ("not"), another one the conjunction ("and"), the last one the disjunction ("or") of its input value(s). Electrical engineers will confirm that it is not unrealistic to assume that such elements can be made, although the elements used in actual hardware design are slightly different.
Can we convince ourselves that any Boolean function with *m* inputs can be realised by a combination of such elements? Yes, as soon as we see that every possible such function can be written as a logical expression in disjunctive normal form. This is one of the results of
• *propositional logic*.

There is nothing mysterious about computer hardware. Once a good *f* has been found, we could make it ourselves by designing a large electronic network of "not", "and", and "or" gates that compute this *f*. But what *is* a good *f*? After some confused discussion we begin to feel that this question has to do with universality *vs.* complexity. We want a simple and cheap machine that nevertheless can do as much as possible.

5.4. Exemplary research questions

By this we have identified the research question at the very root of computer science: is there a machine that can be programmed to do anything we want, and that does it in a cheap and efficient way? For more than half a century, some fundamental results about → universal machines and → undecidable problems have been known. But more than enough questions are still open, and new technologies continuously give rise to new ones. Modern electronic circuits are so small and fast that they can no longer be understood as simple networks of logical gates. Research about → biological computers and → quantum computing is going on. → Decompression of movies requires specialised hardware architectures. And so on and so forth.

It is Friday afternoon. On Monday we started our quest by looking at Internet applications, asking ourselves how they could me made to work. Now we have arrived at the frontier of hardware design, an area where the objects studied are so small that they approach the wave length of light. We have reached the border between computer science and natural science. It is time to conclude the course with some drinks in the garden.

Evaluation

Students and lecturers experienced the course as thorough and fatiguing but at the same time as a great success. We had spent five mornings with each other in intense discussion, and we had the feeling to have reached a lot.
On Friday afternoon the students were asked to evaluate the teaching and learning process and to send in their evaluation by e-mail. About half of them wrote their reports immediately, in a spare hour between the last practical and the reception in the garden.
All reports as they came in can be found in the appendix. Here we give a summary.
The students clearly liked the course and found it a good investment of time. (Only one found it not challenging enough as he already knew everything.) They agree that it –
– gives an impression of what kind of science informatics is,
– allows orientation in a broad area, and
– serves to bring students with different experience to roughly the same level..
Generally, they also liked the form:
– discussions,
– demonstrations,
– technodrama,
– practical,
– the concluding reception with drinks.
There were no clearly negative signals about contents and form.
About the technodramas they were more enthusiastic than expected: the author had feared that they might find this form childish or embarrassing.
The drinks in the end were indeed understood to be functional.
There was also criticism on a number of points:
– the discussions and purpose of the first technodrama were not always sufficiently clear;
– there should definitely have been more coffee breaks;
– the "best" students determined the speed of the discussions;
6. the demonstration of many different applications was not really exciting.

Notwithstanding all problems we encountered, the teachers conclude that such a course indeed is useful and indeed can be given in one week. The most encouraging result of the experiment is that beginning students, prior to any training in computer science, can be helped understand such difficult principles as multiprogramming or bootstrap within a couple of days – provided these subjects are treated at the right level of abstraction. This confirms our conviction that a similar course can be useful for everyone who, being not an ICT professional, would like to get a better understanding of that area.

Literature

Wupper, H. and Meijer, H. *Towards a Taxonomy for Computer Science.* In: *Informatics as a discipline and in other disciplines: what is in common?* Informatics in Higher Education – IFIP WG 3.2 Working Conference, Enschede, Aug. 1997. (Mulder and Van Weert, *ed*) London 1998

Wupper, H. and Meijer, H. *A Taxonomy for Computing Science.* Technical Report CSI-R9713, Computing Science Institute, University of Nijmegen, August 1997 (also available via http://www.cs.kun.nl/~wupper/taxonomy/taxonomy-frame.html)

Appendix

Announcement

Here the official announcement as it appeared in Dutch in the students' guide.

A0. Inleiding informatica

Docenten

Hanno Wupper, Eric Barendsen

Tijdsbesteding

Eén volle werkweek in de eerste week van het eerste semester.

Beschrijving

Deze cursus vormt het begin van de informatica-opleiding en vult de eerste week van het erste semester. Van buiten naar binnen wordt een computernetwerk ontleed tot op het niveau van bits en elektronische schakelingen. Elk van de vijf werkdagen wordt besteed aan een andere schil van de "ui", 's ochtends in de vorm van een college met demonstraties en discussie, waarna de studenten in de middag in een practicum de besproken systeemaspecten zelf onderzoeken.

Lessen

Maandag. Netwerken
Waar komen tegenwoordig overal computers voor? Hoe zijn ze aan elkaar gekoppeld tot een wereldomspannend netwerk? Wat doet dat netwerk allemaal, en welke problemen levert het op?

Dinsdag. Computers
Hoe zit een computer in elkaar? Uit welke onderdelen bestaat hij en op welke manier communiceert hij met de buitenwereld? Hoe werkt een computer eigenlijk, en welke problemen levert dat op?

Woensdag. Bedrijfssystemen
Hoe is het mogelijk dat een computer al zijn complexe taken tegelijk aan kan? Wat gebeurt achter de schermen als een gewone gebruiker met een simpele opdracht een applicatie-programma opstart? Waarom komen de gegevens op de harde schijf niet in de war? Welke programmatuur moet een computer meekrijgen om überhaupt te kunnen worden gebruikt, en welke problemen levert dat op?

Donderdag. Talen
Welke rol speelt taal in het omgaan met computers? Hoe kan een machine taal begrijpen? Hoe kan een machine vertalen van een taal naar een andere? Wat voor talen kunnen machines aan en welke niet, en wat voor problemen levert dat op?

Vrijdag. Processoren
Hoe zit de 'central processing unit' van een computer in elkaar? Wat is een programma, wat is software en hardware? Waaruit bestaat een computer eigenijk? Hoe kan men een machine maken die opdrachten uitvoert die de maker niet kon voorzien, en welke problemen levert dat op?

Doelen

De cursus heeft een aantal doelen:

1. Overzicht over de fenomenen
Waarmee is de informatica eigenlijk bezig? – De deelnemers ervaren een breed, representatief spectrum van fenomenen die de informatica wetenschappelijk bestudeert, opdat ze ze later kunnen herkennen, benoemen en in context kunnen plaatsen.

2. Verwerven van vaardigheden
Hoe gebruikt men computers? – De deelnemers oefenen in het practicum de vaardigheden welke nodig zijn om aan het onderwijs van het eerste jaar goed te kunnen deelnemen.

3. Eerste oriëntatie op het vakgebied als wetenschap
Wat voor wetenschap is de informatica? – Voorbeelden van problemen die zo moeilijk zijn dat ze om een wetenschappelijke aanpak vragen; voorbeelden van onderzoeksgebieden met hun karakteristieke onderzoeksvragen; verwijzing naar de desbetreffende vakken in het curriculum.

4. Eerste oriëntatie op universitair onderwijs
Hoe studeert men informatica in Nijmegen? – Universitair onderwijs lis anders dan het onderwijs op een middelbare school. De deelnemers leren verschillende onderwijs- en leervormen kennen.

Algemene bekwaamheden

De studenten kunnen zich oriënteren in de wereld van IT-toepassingen, de belangrijkste fenomenen benoemen en in context plaatsen.
De studenten kunnen computers en het internet gebruiken.

Inhoudelijke bekwaamheden

De studenten hebben eerste ervaringen in het gebruik van: e-mail, usenet, irc, www, videoconferencing, heterogene netwerken, TCP/IP, spreadsheet, databases, tekstverwerking, de rol van bedrijfssystemen en talen, de personal computer. Ze kunnen hierdoor het gebruik van computers zelfstandig verder leren en/of benoemen wat ze nog moeten leren en welke hulp ze daarbij nodig hebben.

Beginvereisten

n.v.t.

Werkwijze

's Ochtends demonstraties, colleges en discussies; 's middags practicum.

Tentaminering

Nader te bepalen.