Fundamental Ideas of Computer Science

Andreas Schwill
Fachbereich Mathematik/Informatik - Universität Paderborn
D-33095 Paderborn - Germany
email: schwill@uni-paderborn.de

1 Introduction

The current situation of computer science education at the university level is characterized by two aspects:

1. Even with a historical background of more than 40 years computer science stills develops dynamically. Continuously paradigm changes are announced, e.g.
   - from programming in a straight forward manner to structured programming after the software crisis in the sixties, later to logic, functional or object-oriented programming,
   - from sequential execution to parallel execution by autonomous and intelligent processors that communicate,
   - from programming as an art to programming as a science of engineering.

   Since each student will probably face several paradigm changes in future life with much of the respective knowledge becoming obsolete each time, the skills acquired earlier must be robust to latest fashion and enable the student to cope with the changes. Hence, it is necessary that students obtain a sketch of the *fundamental ideas*, *principles*, *methods* and *ways of thinking* of computer science. Only these fundamentals seem to remain valid in the long term and enable students to acquire new concepts successfully during their professional career in that these concepts will often appear to be just further developments or variants of subjects already familiar and then are accessible more easily using ideas learned before.
2. Due to the dynamic evolution of computer science new subjects arise permanently, grow up, become a regular branch and finally will be included into the computer science curriculum. So students have to attend a variety of lectures most covering a very narrow field of computer science and taught in a way as if the fields had nothing in common. A linking structure between these fields is rarely pointed out. Hence, students have to keep in mind all the different details acquired in lectures whereas carefully stressing the *methods and ideas* common to different fields would make things more transparent for them. They would see that things they are taught now are related in that they are just modifications or specializations of a fundamental notion already got to know in another context.

These guidelines without relating to a specific science and meant to improve school education have been developed first by the American psychologist J.S. Bruner in his fundamental work [B60].

In this paper we carry over Bruner's proposals to computer science education at the university level. In section 2 we first sketch the background of Bruner's work. Then we develop criteria for fundamental ideas and propose a general procedure to ascertain fundamental ideas. Another subsection devotes to teaching of fundamental ideas. Finally in section 3 we establish fundamental ideas of computer science and check their fundamentalness according to the criteria developed.

2 Fundamental ideas as an educational principle

As early as 1929 the philosopher and mathematician A.N. Whitehead [W29] proposed to deal in school with few general ideas of universal significance, since the students are helplessly facing a vast amount of details that neither enable them to acquire big ideas nor reveal a connection to everyday thinking.
In 1960 then J.S. Bruner formulated the teaching principle that lessons should predominantly orient towards the **structure** (the so-called **fundamental ideas**) of science.

2.1 Background and motivation

Bruner substantiates his approach as follows: Learning is mainly for preparing us to master our future life more successfully. Since learning under control by a teacher - not mentioning continuing education - is almost finished with the last school year or university semester, changes occuring later in private life, economy and society can only be coped with by **transfering** knowledge earlier acquired to the new situations.

This transfer can be classified with respect to two different aspects:

- If the new situation resembles one already known so that the solution schema of the latter has to be changed or extended only slightly in order to be applicable to the new situation, one speaks of **specific transfer**. Specific transfer relates to relatively local effects and is mostly required if it is upon the short-term application of manual skills within a relatively well-defined area.
- The **nonspecific transfer** however relates to long-term (often life-long) effects. Instead of or in addition to manual skills we learn fundamental notions, principles and ways of thinking (so-called **fundamental ideas**). Moreover we develop attitudes, e.g. to learning itself, to research, to our own achievement, to conjectures, to heuristics, to observations, to solvability of problems etc. Then in some sense problems occuring later in our lives can be considered as special cases and treated with the corresponding solution schemas in transfered form. While specific transfer directly relates something new to something already known of the same logical level, nonspecific transfer includes a metalevel (Fig. 1).

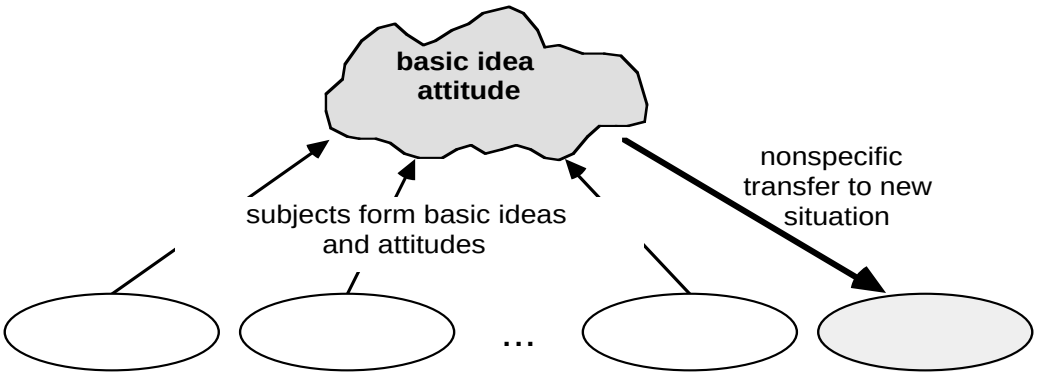


Fig. 1: Nonspecific transfer

During vocational and continuing education specific transfer is dominating. The skills are mainly not taught in a way that forms fundamental ideas in the students. Consequently these skills can usually only be transfered specifically to new problems. On the other side nonspecific transfer dominates the educational process of universities and schools providing general education: Permanent creation, extension and consolidation of knowledge in form of fundamental ideas. This knowledge is not taught in a form to be applied immediately.

Therefore - and now we return to Bruner's request mentioned at the beginning - education has to be oriented mainly towards fundamental ideas. Each subject to be included into lessons and lectures has to be analysed what ideas it is based on. All curricula and teaching methods are to be geared to stress the fundamental ideas of each topic.

Of course, these requests bring up several problems: What are fundamental ideas? How to modify curricula and programs in order to assign a central role to fundamental ideas? Which subjects are most suitable to teach fundamental ideas with respect to level of education? Some of these questions will be treated in the following, first from a general point of view and in section 3 with respect to computer science.

2.2 Fundamental idea: A more precise definition

The central question behind all considerations so far is: What are fundamental ideas? J. S. Bruner himself does not provide an explicit definition of this notion. Instead he gives many examples of fundamental ideas and so leaves it to the reader to obtain an intuitive idea of what this term may denote. Just a few, yet important hints are scattered over his work, e.g.

> „...what is meant by ‚fundamental' in this sense is precisely that an idea has wide as well as powerful applicability" (p. 18).

Hence, fundamental ideas are widely applicable in many contexts and organize and

integrate a wealth of phenomena. We call this property the **horizontal criterion** having the following illustration in mind: A horizontal axis penetrating a great number of application fields (Fig. 2).

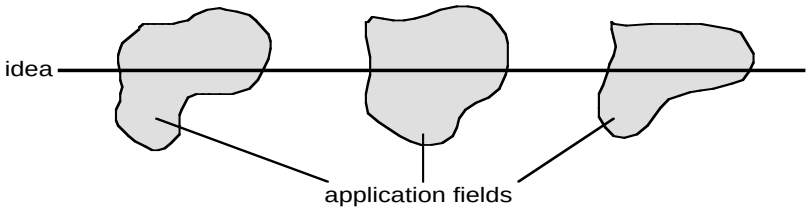


Fig. 2: Illustration of the horizontal criterion

Bruner's statement that, if earlier learning is to make later learning easier, then there must be a general principle that reveals the connections between things confronted with earlier resp. later, shows in combination with his famous thesis

> „that any subject can be taught effectively in some intellectually honest form to any child at any stage of development." (p. 33),

that fundamental ideas structure the topics of a field also vertically: A fundamental idea can be taught on almost every level of understanding, i.e. on the level of a primary school as well as on the university level (**vertical criterion**). Presentations differ only by level of detail and formalization. Formulated the other way round (after R. Fischer [F84]): Topics that cannot be taught to students in primary school cannot be fundamental ideas.

Hence, fundamental ideas must be acquirable in early stages of the development of the human brain. So they cannot be made as objective as possible in some abstract sense or defined as a law of nature independent of man. Nevertheless we shall try to present an objective catalog in section 3, which however is objective only insofar as a number of people might accept the notions as fundamental ideas of computer science.

Vividly speaking we can subdivide the application field of a fundamental idea into different levels of growing intellectual demands. Thus, a fundamental idea is represented by a vertical axis intersecting each level (Fig. 3).

The vertical criterion is of particular relevance for lessons: An idea satisfying this criterion can serve as a guideline for lessons and lectures on every level of the entire educational process. In connection with the spiral principle we return to this later.

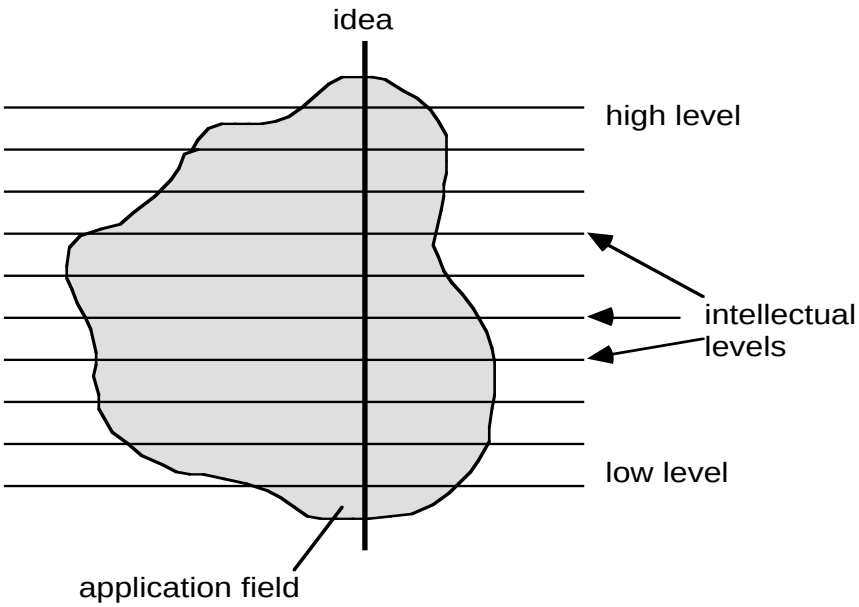


Fig. 3: Illustration of the vertical criterion

This finishes the review of Bruner's criteria for fundamental ideas. In the following we add the opinions of two different authors which, although relating to mathematics, also contain general remarks.

A. Schreiber's [S83] contribution, mainly in line with Bruner's philosophy, consists of a list of handy clues for fundamental ideas of mathematics, namely:

- **Width**, i.e. logical generality. What he probably means is that an idea has width if it is versatile to a certain degree and leaves some freedom for interpretations and formalizations. Hence, this property rules out exactly defined algorithms, axioms or laws. *Example:* The law of commutativity a+b=b+a is not an idea, since it is lacking generality. *Invariance* may be considered an idea with width instead, which subsumes the law of commutativity (invariance against change of operands) as well as many other phenomena in physics, chemistry and other sciences. Likewise the famous equality $E=mc^2$ or the quicksort algorithm are not fundamental ideas.
- **Richness**, i.e. a wealth of applicability and relevance. This criterion almost coincides with our horizontal criterion.
- **Sense**, i.e. embodiment in everyday life. This criterion - we call it the **criterion of sense** - extends Bruner's criteria. A fundamental idea according to Schreiber still belongs to the sphere of everyday thinking its context being pretheoretical and unscientific. Only a precise definition turns an idea into an exact notion. Cf. the quotation of Whitehead at the beginning of section 2.

Example: The relation sketched above between an idea with „sense“ and a notion exists between the idea of „reversibility“ and the notion of „inverse function“. While „inverse function“ is a pure mathematical notion having no everyday life meaning, the idea of reversibility can be pointed out in many situations in everyday life.

In his paper F. Schweiger [S82] determines fundamental ideas of analysis where he defines fundamental ideas to be a

> „sheaf of actions, strategies or techniques tied together by loose analogy or transfer which
> (1) are demonstrable in the historical development of mathematics,
> (2) appear to be sound to structure curricula vertically,
> (3) seem to be suitable ideas concerning the question „What is mathematics?“,
> (4) could make mathematical lessons more flexible and more transparent. Furthermore
> (5) an embodiment in everyday language and thinking, so to speak a corresponding archetype with respect to thinking, speaking or acting seems necessary.

Schweiger's opinions differ from Bruner's and Schreiber's in several ways. While listing the vertical criterion (item (2)), he does not mention the as important horizontal criterion. On the other hand Schweiger stresses (as does Schreiber) the relation of fundamental ideas to everyday life (item (5)).

Two other aspects are new: the historical one (item (1)) and the philosophical one (item 3)). The historical aspect - we speak of the **criterion of time** henceforth - is important for two reasons. On the one hand it gives a clue how to find fundamental ideas: By observation of the historical development of scientific notions, concepts and structures. On the other hand it indicates that fundamental ideas of a science will be relevant for a longer period of time. This property has also been observed by J. Nievergelt [N90] when expressing his „quest for classics“:

> „How do we recognize ideas of long lasting-value among the crowd of fads? The ‚test of time' is the most obvious selector. Other things being equal, ideas that have impressed our predecessors are more likely to continue to impress our successors than our latest discoveries will.“ (p. 5)

The philosophical aspect (as well as item (4) of Schweiger's list) seems to be rather a criterion for but an advantage of fundamental ideas. For having structured a science by a list of fundamental ideas one also has a philosophically sound basis of the science, knows its essence and can differentiate it from other sciences.

Now we wish to summarize the above remarks by a definition of our own which combines the four criteria elaborated above.

„Definition“:

A **fundamental idea** with respect to some domain (e.g. a science or a branch) is a schema for thinking, acting, describing or explaining which

(1) is applicable or observable in multiple ways in different areas (of the domain) (**horizontal criterion**),

(2) may be demonstrated and taught on every intellectual level (**vertical criterion**),

(3) can be clearly observed in the historical development (of the domain) and will be relevant in the longer term (**criterion of time**),

(4) is related to everyday language and thinking (**criterion of sense**).

Up to now fundamental ideas have been proposed mainly for mathematics and some of its branches, namely theory of probability, analysis, linear algebra, numerical mathematics, group theory and geometry [F76,H81,H75,K81,M80,S82,T79]. Other approaches concern physics, chemistry and biology [S70,S81,G77].

2.3 How to determine fundamental ideas?

There are very few comments in literature concerning this question, anyway it is a difficult problem (according to Bruner) since it requires a comprehensive overview of a science to prove single ideas fundamental, to obtain a *complete* collection of fundamental ideas of the science, to show their independency or to find hierarchies between them. For the moment the following program which abstracts from the contents of a science to its ideas seems of practical use only:

Step 1: Analyze the concrete contents of a science and determine relationships and analogies between its subjects (horizontal criterion) as well as between different intellectual levels (vertical criterion). This gives a first collection of fundamental ideas.

Step 2: Revise and improve this list by checking whether each idea has a meaning and can be found in everyday life (criterion of sense).

Step 3: Try to review the historical development of each idea. This possibly gives other viewpoints and stabilizes the collection of ideas. Also consider the suggestion of Nievergelt (section 2.2).

Step 4: Finally the list of ideas is tuned according to the following questions: Do the ideas have a similar level of abstraction? Is it possible to structure or group the ideas somehow? Are there any hierarchical or network dependencies between them? Are the ideas „linearly independent“?

After a possibly iterative execution of this procedure one may obtain a stable collection of fundamental ideas. However one has to keep in mind that every list so obtained is affected by the person performing this procedure, because none of the notions and conditions involved is nor can be formally defined, i.e. always leaves room for interpretations. Furthermore there is no criterion to prove the completeness of a collection of ideas. At best this property is gained either by discussion and regular use of the list within the scientific community or by orienting education towards ideas in the sense of Bruner.

2.4 How to teach fundamental ideas?

The most important contribution stems from Bruner himself when he requires that lessons oriented towards fundamental ideas have to be organized according to the **spiral principle** which he describes as follows:

> „The early teaching of science, mathematics, social studies, and literature should be designed to teach these subjects with scrupulous intellectual honesty, but with an emphasis upon the intuitive grasp of ideas and upon the use of these basic ideas. A curriculum as it develops should revisit these basic ideas repeatedly, building upon them until the student has grasped the full formal apparatus that goes with them.“ (p. 13).

When analyzing the parts of this description in more detail one recognizes three subprinciples which in a way form the basis of the spiral principle:

- **Principle of extendibility**: A subject has to be selected and treated at a specific place within the curriculum in such a way that it can be extended at a higher level. One has to avoid approaches (also by teaching half-truths) that were chosen for didactic reasons only but later require a change of views and a revision of statements.
- **Principle of prefiguration of notions**: The *symbolic* demonstration of notions or concepts and its structural analysis has to be prepared already on a lower level by pictures (*iconic*) and actions (*enactive*). That means: before a notion can be analyzed theoretically in full detail, it should be first put into use so that students obtain an intuitive idea of it.
- **Principle of anticipated learning**: The treatment of a scientific field should not be delayed until a complete and detailed analysis seems possible but has to be initiated before on a lower level.

2.5 Advantages of Bruner’s conception

Bruner himself mentions the following advantages:

- A subject is more comprehensible if the student understands its fundamental principles.
- Fundamental ideas condense information by organizing uncoherent details into a linking structure which will be kept in mind for a longer time. Details can be reconstructed from this structure more easily.
- Fundamental ideas support non-specific transfer. Their generality allows many problems to be treated as special cases.
- Since fundamental ideas structure the subject vertically they reduce the lag between current research findings and what is taught in schools or lower levels of undergraduate studies. This expresses Bruner’s conviction that

  > „... intellectual activity is the same anywhere, whether the person is a third grader or a research scientist.“ [B60].

  So the activities of a scientist and a student in primary school do not differ in essence, since both apply the same fundamental ideas, yet on a different level.

Further advantages:

- While fundamental ideas remain modern even in the longer term (criterion of time), details become antiquated very early. This holds in particular for computer science with its dynamic evolution so that Bruner's conception appears most powerful here.
- By now there is no branch called philosophy of computer science. Thus, a collection of fundamental ideas may serve as a first approach in order to determine the essence of computer science and to dissociate it from other sciences.

3 Fundamental ideas of computer science

In order to develop a collection of fundamental ideas of computer science we mainly follow the plan proposed in section 2.3. Of course we cannot perform each of the four steps in detail, the principle may suffice here. Also we will not check each of the ideas we obtain whether it satisfies the criteria mentioned in the definition. For it some examples might be enough.

Now consider the first step: Analysis of concrete contents and determination of relations and analogies between its subjects. What are the concrete contents of computer science to analyze here?
A central purpose of computer science is to investigate the software development process in its broadest sense and to provide methods for it. Consequently it seems reasonable to analyze this process for fundamental ideas as is done in the following section.

3.1 Software development

The basis for research in software development is the software life cycle whose standard representation is given in Fig. 4 [CS90].

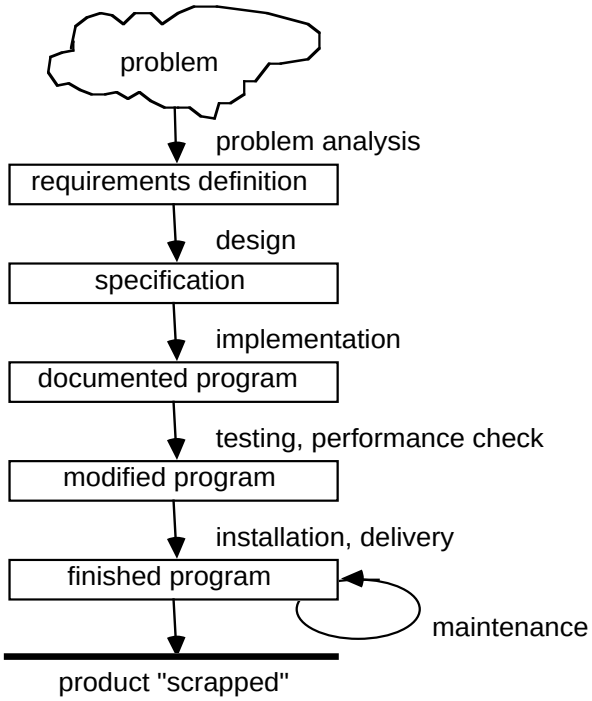


Fig. 4: Software life cycle

Let us consider the stages in more detail.

Problem analysis. In this phase the problem to be solved and all relevant aspects of the environment in which the proposed software system would be used are formally stated. The phase is divided into four steps:

- Analysis of the current situation: Study and description of the existing system by analyzing its components, their functions and interrelationships.
- Target concept: Definition of demands on the software to be developed by specifying user model, underlying hardware, user interface etc.
- Feasibility study: This study gives information whether the ideas about the software product are realizable at all in their desired form, whether they are feasible in principle (e.g. are there requirements that describe non-computable problems?) and economically attractive. The outcome of this study either leads to an abort or to a revision or to carrying out the project.
- Project planning: Development of time flow charts, allocation of staff to teams etc.

The results of the entire problem analysis phase is recorded in a document called requirements definition which becomes part of the contract between client and producer.

The central idea in this stage is *structured dissection:* The vague ideas of the client are written down in a precise and structured way. The structure of the entire system given is revealed by breaking down the system into its components and determining their

interrelations. Usually such an analysis is subdivided into several steps by first dissecting the system roughly into several parts which then are further refined until a sufficient grade of detail is achieved. This process gives a *hierarchy* of different levels of abstraction. The feasibility study deals among others with problems of computability and complexity in a theoretical sense. Corresponding ideas are e.g. *reduction*, *diagonalization*.

Design. While the problem analysis stage describes properties of the software product without paying regard to how they will be implemented, the programmers develop in the design phase a model of the system that satisfies the requirements when compiled into a program. To this end the entire complex system is split up into easily comprehensible units whose functions and interfaces are described in detail. A standard method is hierarchical modularization which has two different pure approaches called top-down and bottom-up. In order to obtain a structure of the modules as clear as possible, we must ensure that

- each module uses the functions from a minimum number of other modules,
- the interface of each module is as small as possible,
- each module hides as much information as possible about its internal structure (information hiding).

The design stage results in a specification that contains for each module its function, interface, comments on applicability as well as an overview of all modules and their interrelations.

In the center of the design stage is the idea of *modularization* with its two mouldings (*top-down* and *bottom-up*). In order to be able to modularize at all, certain conditions have to be satisfied: The function of each module must be formally defined by a *specification language* that provides a *parameter concept* and supports *information hiding*.

Implementation. Development of an executable program whose input/output-behaviour satisfies the requirements specification. In this stage the selection of a programming language is particularly important. The main activity concerns the implementation of the modules. In order to be able to test and change modules more easily later, one has to keep the following guidelines among others:

- structured programming,
- clearly defined interfaces using parametrization,
- use of semantically simple elements of the programming language.

It is important that the modules as developed in the design stage are reflected in the program as easily comprehensible units. Implementation results in a fully documented program.

The central part of the implementation stage is the idea of *algorithmization* (*dissection* into single steps) and the subsequent conversion of an algorithm into an *executable* program

of a *programming language* consisting of *control structures* (*sequence*, *loop*, *alternative* etc.) and *data structures* (*aggregation*, *generalization* etc.).
Depending on the problem to be solved algorithms may be selected on the basis of different fundamental patterns, often called paradigms, such as *divide-and-conquer*, *branch-and-bound* etc. Implementation itself has to follow the guidelines of *structured programming*. For the modules to be visible in the program the programming language must at least contain a *block* and a *parameter concept* as well as means for graphical or linguistic representation of *hierarchies* (parantheses, begin ... end, indentation).

Testing. According to the requirements specification the program's input/output-behaviour is checked by a combination of testing and verification. It all starts with the module test. The function of each module is checked with respect to its specification. After putting all tested and verified modules together integration test ist performed. It follows the installation test and finally the delivery test.

Main idea in this stage is the *quality control*, i.e. the analysis of the finished program or parts of it for *correctness*. All components of the program are checked either formally or by test cases for *partial* and *total correctness*. If concurrent programs are involved, ideas like *consistency* and *fairness* also play an important role.

Performance check. After checking for correctness the program's performance is assessed.

As above the idea of quality control, in particular the idea of *complexity*, is in the center of this stage. Fundamental ideas related to this are among others the notions of *order* or of *worst case complexity*.

The remaining stages of the software life cycle are installation, delivery to the client and maintenance of the product. New ideas not mentioned yet do not arise here.

3.2 A collection of fundamental ideas

Among the ideas mentioned in section 3.1 there are three that play an important role since they dominate all stages of software development. The idea of *algorithmization* has already been mentioned explicitly, the other two arise implicitly only, the idea of *language* and the idea of *structured dissection*. Both ideas will be analyzed in more detail now as they give hints for several other fundamental ideas.

Language. Not only for programming (programming languages), for specification (specification languages), for verification (logic calculi), in data bases (query languages), in operating systems (command languages) the idea of language plays an important role,

but there seems to be a general trend in computer science to formulate any facts by a language. This even holds in fields where at first sight there seems to be no plain relation to a linguistic representation, e.g. in VLSI design or in design of logical circuits. This approach has the following advantage: On the one hand it standardizes the view of facts, since every problem can be considered as a problem upon words now; on the other hand manipulation, especially translation, of languages and words has been successfully investigated in the past.

Example: A computer architecture is often modelled by a multilevel machine [T84] (Fig. 5). Each level is assigned a language for describing objects and operations of this level in a suitable manner. The user usually views the top level. Each operation he sends to the system is stepwise transformed into the lower language level and finally into the hardware.

In close relation to languages there are obviously the ideas of *syntax* and *semantics*, furthermore the different approaches to transform words of one language into words of another language keeping their semantics, e.g. the ideas of *translation*, *interpretation*, *operational extension*.

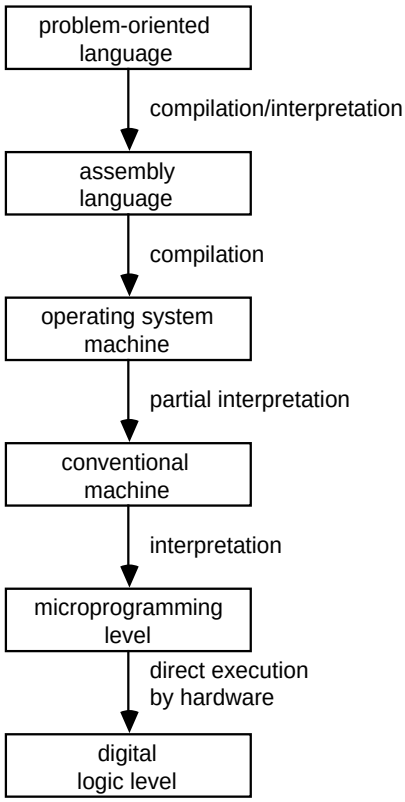


Fig. 5: Computer architecture as a multilevel machine

Structured dissection. Dissection appears for example

- in problem analysis when the present system is iteratively split up into components or when teams are created,
- in hierarchical modularization during the design stage,
- during implementation stage when processes are broken down into single steps (cf. algorithmization) or when a problem is subdivided into smaller ones (divide-and-conquer)
- during software life cycle itself where the development process is subdivided into six stages each of which comprises several substages.

Obviously we can distinguish two aspects in the idea of dissection, a vertical aspect made concrete by *hierarchization* (Fig. 6) and a horizontal one made concrete by *modularization* (Fig. 7). *Hierarchical modularization* is obtained then by merging these two aspects (Fig. 8).

The idea of hierarchization can also be observed in many different contexts: level-oriented models of computer architecture (see above), language hierarchies (main example is the Chomsky hierarchy), machine models, complexity and computability classes, virtual machines, ISO-OSI reference model.

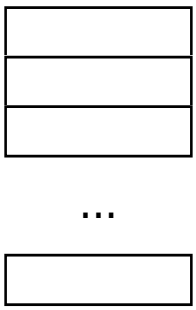


Fig. 6: Hierarchization

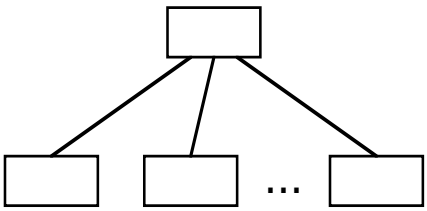


Fig. 7: Modularization

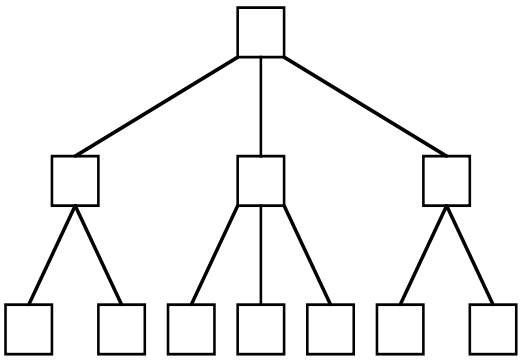


Fig. 8: Hierarchical modularization

Another important aspect of dissection has not been mentioned yet: Obviously every dissection procedure comes to an end some time, at the latest if an atomic level is achieved.

Example: In a divide-and-conquer algorithm dissection stops when a problem is achieved whose solution is obvious.
Hierarchical modularization stops as soon as a module specification is achieved that can be transformed into a single statement of the underlying programming language.

This observation leads us to the fundamental idea of a generating system, that we call *orthogonalization*, inspired by a similar operation for vectors in linear algebra.

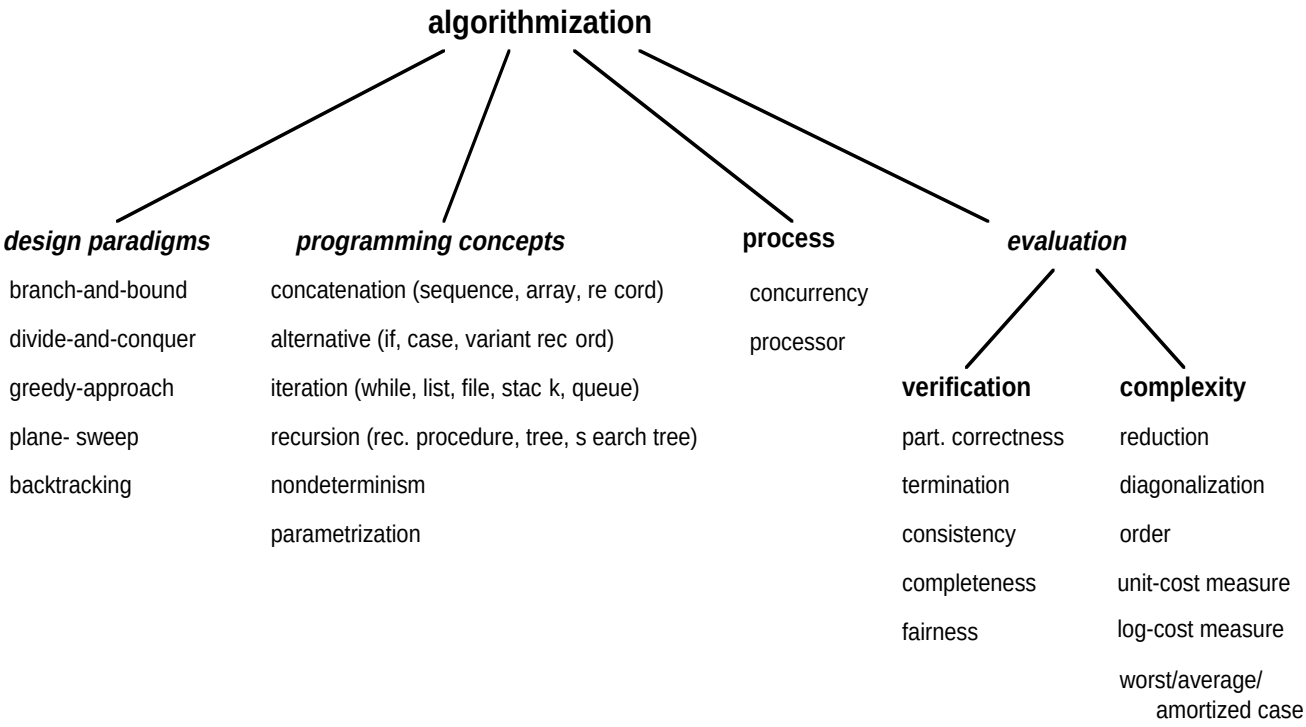
Orthogonalization denotes an operation on a domain that gives a small as possible number of basic elements and a set of operations on this basis such that every object in the domain can be generated from the basis by finite application of operations. That orthogonalization is in fact a fundamental idea can be seen from its multiple applications within and outside of computer science (cf. horizontal criterion and criterion of sense):

- programming languages. A fundamental principle of ALGOL68 has been its orthogonal design: There are few basic elements for defining data structures and control structures that may be combined in an arbitrary way. Arrays as well as other data types for instance may be used as result types of functions. PASCAL is much more restrictive here and hence cannot be called orthogonal.
- imperative programming languages. The structures assignment, concatenation and while-loop form a basis for all control structures. Every other statement may be simulated by these three basic ones.
- functional programming languages. The basic operations in languages that are based on the λ-calculus are abstraction (i.e. definition of a function with parameters), application (i.e. function call with actual parameters) and parameter substitution.
- machines. The universal Turing machine is the (one element) basis of the class of all Turing machines.
- primitive recursive and μ-recursive functions. There is a set of basic functions (e.g. constant function 1, successor function) and a number of operations (e.g. composition, substitution, μ-operator) with which every other primitive recursive resp. μ-recursive function can be generated.
- formal languages. The Dyck language is in some sense a basis of the class of context-free languages (theorem of Chomsky-Schützenberger).
- Boolean functions. AND, OR and NOT form a basis of all Boolean functions. NAND is also a basis.
- assembly of cars with modular parts,
- wall-to-wall cupboards built with standardized elements
- houses built with prefabricated parts
- DNS consisting of four basic elements.

For proving that a system is not orthogonal the idea of *emulation* is often used: Given a basis if one of its elements can be expressed by means of the others the basis is not orthogonal.

After these considerations we can now establish our complete catalog of fundamental ideas of computer science. It contains all ideas already mentioned grouped by subjects and structured hierarchically. Some new ideas polish the groups. „Master ideas“ are algorithmization, structured dissection and language (Fig. 9). Note that names written in italics have been added for systematization only and denote groups of ideas but are not ideas themselves.

Obviously, the assignment of ideas to master ideas is not unique, since some ideas combine different aspects. The divide-and-conquer approach, for instance, contains both an algorithmic and a dissection aspect. Then we have assigned divide-and-conquer to algorithmization since the dynamic aspect (process aspect) prevails. On the contrary dissection stresses the static aspect, i.e. the result of the dissection process and not the way the result is achieved. Furthermore several ideas occur multiple times in different contexts in the catalog. Reduction and transformation, for instance, denote translation processes, translation on the other hand appears again as an idea for implementing hierarchies. So it can be seen that ideas are intertwined in many ways, an exact separation and assignment is hardly possible.



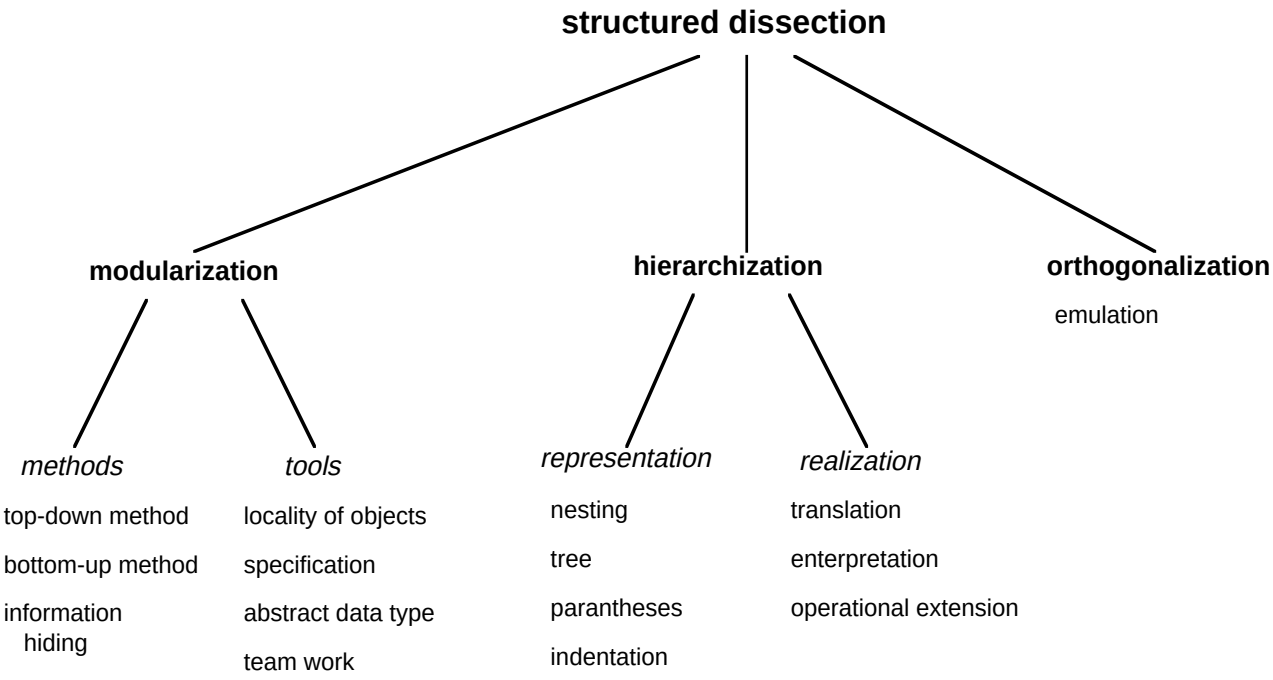


Fig. 9: Fundamental ideas of computer science

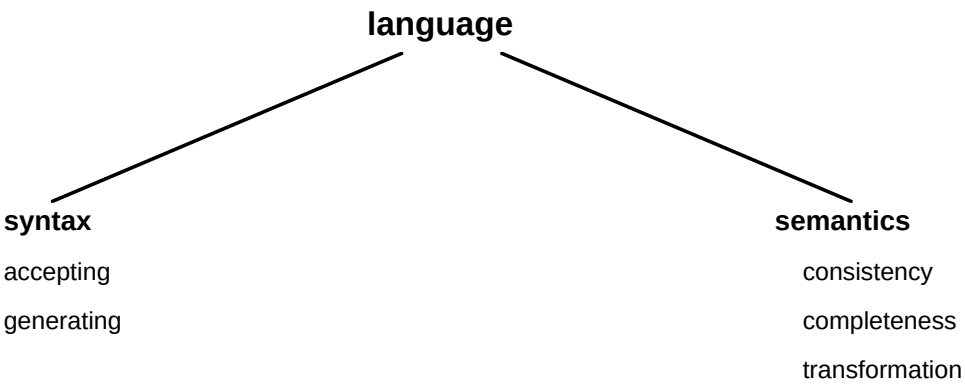


Fig. 9: Fundamental ideas of computer science (continued)

3.3 Verification of fundamentalness

Now we wish to check some ideas for fundamentalness based on the four criteria in the definition. The master ideas may easily be proved fundamental. The criterion of sense and the horizontal criterion has been roughly checked already in section 3.2

Verifying the vertical criterion we restrict ourselves to sketching subjects for lessons in primary school (P), grades 5-9 (S1) and grade 9 and above (S2) respectively of secondary

school that may be used to teach selected ideas on the corresponding intellectual level. The criterion of time is treated at the end of this section.

Divide-and-conquer approach.

Horizontal criterion: Used in sorting and searching algorithms, in all problems based on trees, in matrix multiplication, in computational geometry, in the separation of graphs.

Vertical criterion:

(P) A child may sort a stack of paper cards by size by dividing the stack, giving the parts to its class-mates and merging the sorted stacks he is given back. Numbers may be guessed by binary searching.

(S1) Algorithms in computational geometry, e.g. for computing the convex hull, may be used to deepen the knowledge.

(S2) Complexity considerations for general divide-and-conquer algorihms; establishing and solving a recurrence relation for the runtime.

Criterion of sense: The approach appears in everyday life in many situations, e.g. in all forms of hierarchically organized divisions of labour or in different search procedures: A child has lost a toy. Other children help him. Each child looks for the toy in another area.

Worst case-analysis.

Horizontal criterion: Worst case analysis is done for algorihms with respect to time and space, for time flow-charts in projects (e.g. critical path method), for error estimation in floating point arithmetic or for probabilistic algorithms.

Vertical criterion:

(P) Worst case considerations can start with questions like: How long does it take to get to school in the worst case, if the bus is late, if all traffic lights are red, if the roads are icy,...? Or: How many questions are necessary in order to find out a number guessed by a class-mate using binary search?

(S1) A more formal approach may follow at this level, e.g. by relating the runtime to the length of the input and determining the worst case for each input length.

(S2) Formal definition of worst case runtime and proof of lower bounds.

Criterion of sense: In everyday life worst case considerations often appear in risk estimations, e.g. in financing a home (What mortgage am I able to pay for even in the worst case (unemployment)?), in defining the greatest accident that can occur in a nuclear power station, in defining the safety distance between two cars as being the distance that allows to stop even if the car in front makes a full braking.

Abstract data type.

Horizontal criterion: Used in all forms of specifying objects that stress operations and their properties without relating to an implementation.

Vertical criterion:

(P) The natural numbers may be defined as an abstract data type with constants 1 and operations +1 and -1. The same holds for the blocks world: On a table there is a number of blocks that may be piled up. Operations are putting one block onto another one and testing whether a block lies on another one or whether it lies immediately on the table. Is it possible to establish any situation by these operations (idea of completeness)?

(S1) The two examples mentioned above may be made more precise here. Problems concerning consistency and completeness of an abstract data type may follow. Which laws hold for the operations in the blocks world?

(S2) On this level a formal notation for abstract data types may be introduced using more complex examples (stack, queue, file). Considerations on implementation may follow.

Criterion of sense: An approach similar to abstract data types appears in constructing a machine by specifying its behaviour. In particular, this principle is relevant in open calls for tenders that only specify the „abstract“ behaviour of the object without mentioning any „implementations“ specific for an eligible producer.

Now it remains to show how ideas occur in the historical development of computer science (criterion of time). Some important stages of computer science since 1950 are illustrated in Figs. 10 through 12.

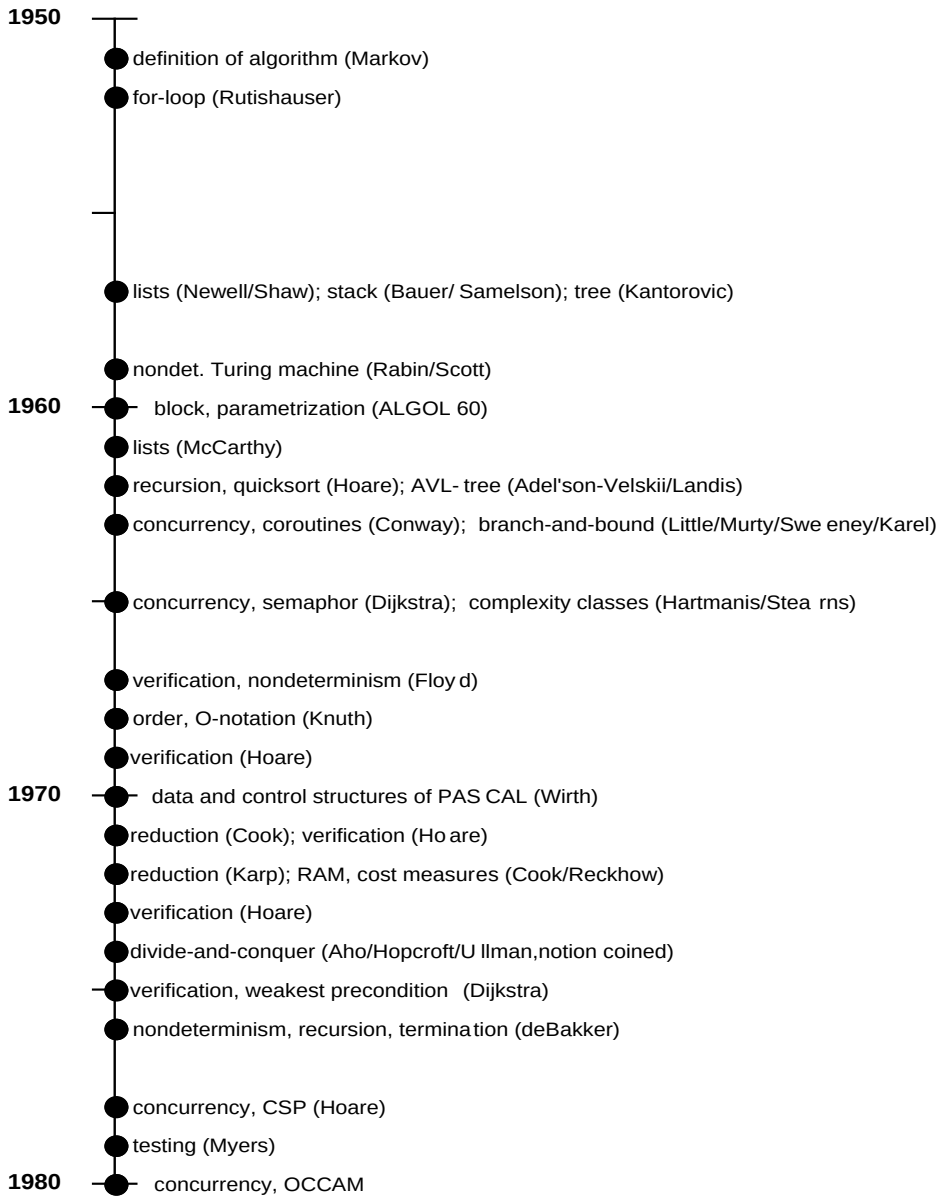


Fig. 10: Historical development of algorithmization and corresponding ideas

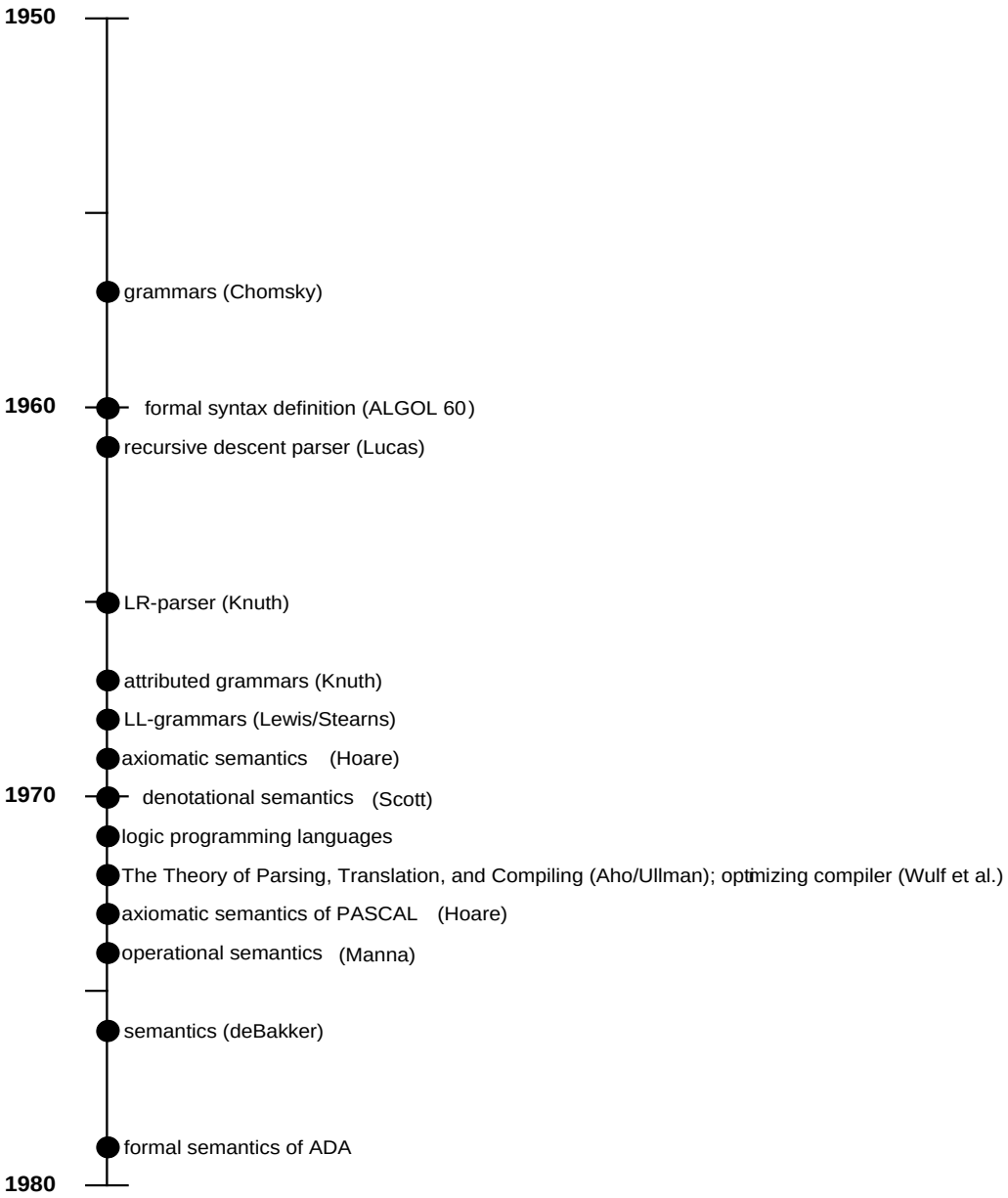


Fig. 11: Historical development of language and corresponding ideas

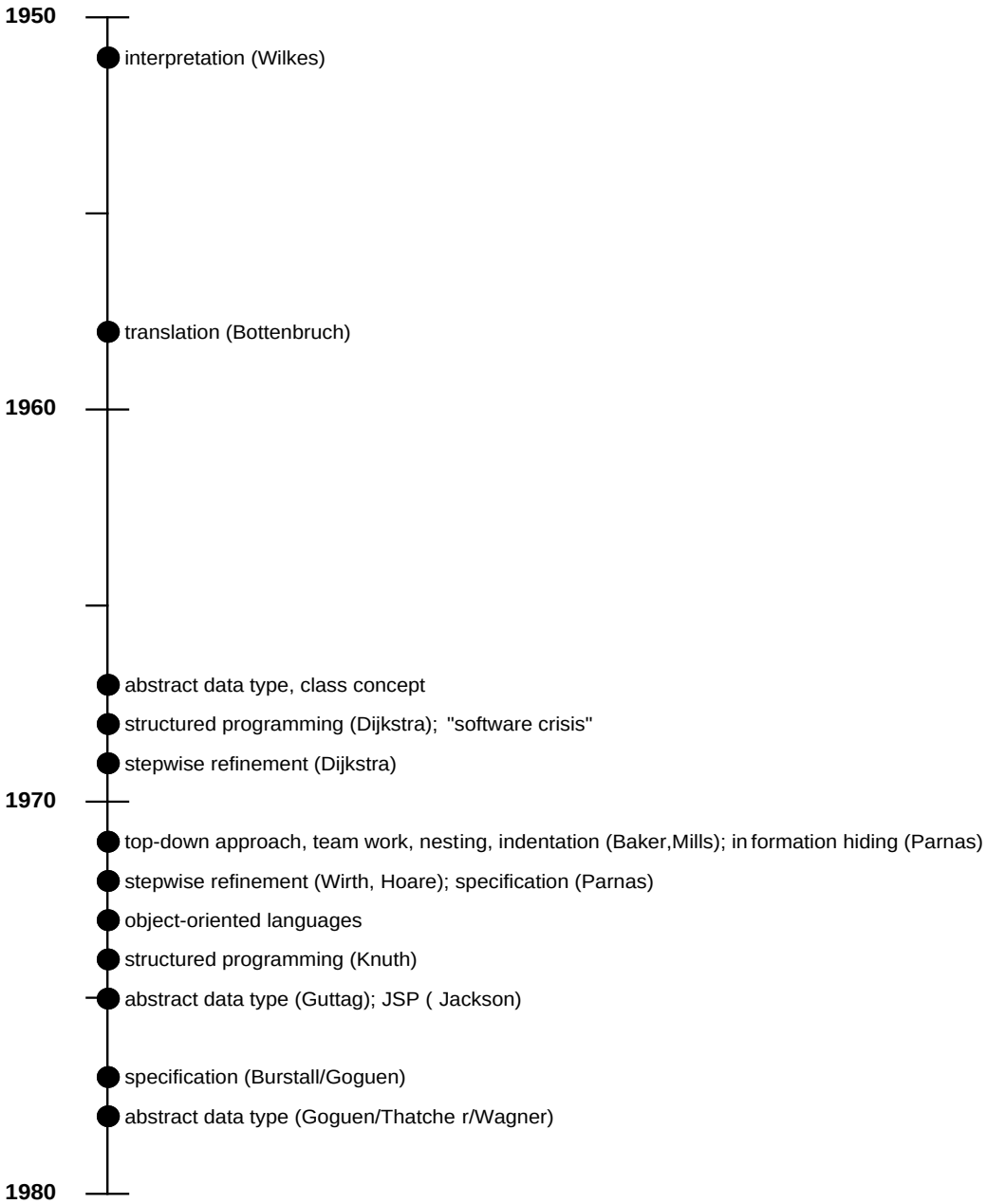


Fig. 12: Historical development of dissection and corresponding ideas

4 Conclusions

In this paper we have transfered J.S. Bruner's principle of orienting education towards fundamental ideas to computer science and brought it up for discussion. According to the considerations so far this principle seems to be a useful approach to structure computer science education. In order to support this thesis several advanced investigations would

be helpful:

1) Development of curricula and programs for computer science education that stress fundamental ideas.
2) Elaboration of suitable examples which highlight certain ideas.
3) Even if the presented catalog of ideas is based on an objective analysis of activities and methods, it is undoubtedly influenced by the author. Hence, further discussion is necessary in order to refine and substantiate the catalog within the scientific community.

Finally, we wish to illustrate by three visionary examples the difference between traditional (subject-oriented) education and an idea-oriented education as proposed here:

traditional (subject-oriented) education	idea-oriented education in pure form
Lectures are structured by subjects. Each subject is dealt with until all details are taught.	Lectures are structured by ideas. Each idea is dealt with until all details (e.g. applicability) are taught.
Lectures concerning a large field cover several semesters, e.g. we have lectures called databases I, II, III.	Lectures concerning a big idea (e.g. a master idea) cover several semesters, e.g. we have lectures called divide-and-conquer I, II or hierarchization I, II, III, IV.
There are professorships and chairs for subjects like databases, complexity theory or operating systems.	There are professorships and chairs for ideas like divide and conquer, specification or orthogonalization, whose research area covers all applications of the resp. idea.

References

[B60] Bruner, J.S.: „The process of education“, Cambridge Mass. 1960

[CS90] Claus, V.; Schwill, A.: „Encyclopaedia of information technology, Ellis Horwood 1990

[F76] Fischer, R.: „Fundamentale Ideen bei den reellen Funktionen“, Zentralblatt für Didaktik der Mathematik 3 (1976) 185-192

[F84] Fischer, R.: „Unterricht als Prozeß von der Befreiung vom Gegenstand - Visionen eines neuen Mathematikunterrichts“, J. für Mathematik-Didaktik 1 (1984) 51-85

[G77] Gärtner, H.: „Lehrplan Biologie - Analyse und Konstruktion“, Sample Verlag 1977

[H81] Halmos, P.R.: „Does mathematics have elements?“, The Mathematical Intelligencer 3 (1981) 147-153

[H75] Heitele, D.: „An epistemological view on fundamental stochastic ideas“, Educational Studies in Mathematics 6 (1975) 187-205

[K81] Klika, M.: „Fundamentale Ideen der Analysis“, mathematica didactica 4 (1981) 1-31, Sonderheft

[M80] Müller, M.W.: „Fundamentale Ideen der Numerischen Mathematik“, Beitr. zum Mathematikunterricht (1980) 238-245

[N90] Nievergelt, J.: „Computer science for teachers: A quest for classics, and how to present them“, Proc. of the 3rd Intern. Conference on Computer Assisted Learning (1990) 2-15, LNCS 438

[S81] Schmidt, H.J.: „Fachdidaktische Grundlagen des Chemieunterrichts“, Vieweg Verlag 1981

[S83] Schreiber, A.: „Bemerkungen zur Rolle universeller Ideen im mathematischen Denken“, mathematica didactica 6 (1983) 65-76

[S82] Schweiger, F.: „Fundamentale Ideen der Analysis und handlungsorientierter Unterricht“, Beitr. zum Mathematikunterricht (1982) 103-111

[S70] Spreckelsen, K.: „Strukturelemente der Physik als Grundlage ihrer Didaktik“, Naturwiss. im Unterr. 18 (1970) 418-424
[T84] Tanenbaum, A.S.: „Structured computer organization“, Prentice-Hall 1984
[T79] Tietze, U.-P.: „Fundamentale Ideen der linearen Algebra und analytischen Geometrie- Aspekte der Curriculumsentwicklung im MU der SII“, mathematica didactica 2 (1979) 137-163
[W29] Whitehead, A.N.: „The mathematics curriculum“, in: The Aims of Education, MacMillan 1929

Note: This paper is a modified version of a paper that originally appeared (in German) in Zentralblatt für Didaktik der Mathematik No. 1 (1993).