STRATEGIES FOR TEACHING UNDERGRADUATE PROGRAMMING

by

Guy Tremblay

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

Research Report CS-85-13
May 1985

Strategies For Teaching Undergraduate Programming

by

Guy Tremblay

An essay
presented to the University of Waterloo
in partial fulfillment of the
requirements for the degree of
Master of Mathematics
in
Computer Science

Supervisor: Marlene Jones Colbourn

Waterloo, Ontario, December 1984

Acknowledgements

Special thanks to Marlene Colbourn, who officially supervised this essay. She read it so many times, correcting the writing style and the many english mispelling. She was always confident that I would finish in time, even though the first results were late to come by. Thanks a lot Marlene.

I also wish to acknowledge the financial support (postgraduate scholarship) which I received from NSERC.

Finally, I would like to thank Nancy, Rebecca, Paul, Susan, Theodora and all those very special people, especially you, Don and Caroline, who gave me space to be and to grow, offering me warmth and support. What you gave me is more important than any Master's degree.

Table of Contents

1. Introduction

As computers play a more pronounced role in our society, more and more people will require some basic knowledge regarding computers, including programming skills. However, many such users will not need to be programming experts. Therefore, one important educational issue to be addressed here is how to facilitate learning of computer-related skills, particularly programming skills. As Soloway points out:

> The need for the public to be literate in computing is rapidly being recognized. One aspect of such literacy is programming. While we do not believe that everyone needs to become a professional programmer, it is increasingly important to be able to describe to the computer how it is suppose to realize one's intentions. [42-p.853]

It is this topic which we address here. Hence, it is related to the emerging field of software psychology, which is "the study of human performance in using computers and information systems." [36-p.3]

The basic idea behind software psychology is that one of the most important aspects of computer science is one which has been ignored for a long time: the human aspect. Of course, this can easily be explained. In the first years of computer science, all the problems were mainly problems of hardware. The cost of programming, of human work in software was negligible compared to that of hardware. As Dijkstra said:

> To put it quite bluntly, as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem. [4-p.861]

The importance of software design and correctness has become, in the last few years, very important. The software psychology approach goes even further by saying that it is people who design the software, who use this software in every day life. So what we should do is study how people perform those tasks. This is expressed by two pioneers in the field of software psychology, Weinberg and Shneiderman:

> So rather than be concerned so much about that computer operating system, the reader who has really been touched by this book will start to work on the operating system he carries around in his own central processing unit - his head. This will be his reward. [46-pp.277,278]

> By recognizing that computers are merely tools and that humans provide the creative energy to keep them moving, we focus attention on human effectiveness rather than machine efficiency. [36-p.10]

The question then, is how should we study those human aspects? Software psychology claims that we should use psychology:

> Understanding of human skills and capacity to design effective computer systems can be improved by application of the techniques of experimental psychology; the analysis of cognitive and perceptual processes; the methods of social, personnel and industrial psychology, and the theories of psycholinguistics. [36-p.3]

This approach is in contradiction to the more traditional computer science approach. For many years, people have made claims about how their language was easier to learn or to use, how their design methodology lead to better programs, programs easier to understand, to modify, etc. Generally, those claims are based on intuitions or personal experience of the designer, rather than experimental evidence. One goal of software psychology is to be more rigorous:

> Alternatively, one might base the design of a L/E/M [1], at least in part on psychological grounds, i.e., develop a theory of how and why people work the way they do, carrying out empirical tests of such hypotheses, and derive design implications from this principled basis. [43-p.193]

Of course, an aspect which is very important when people start dealing with computers is how they learn to use them. This is one of the issues we address here: how people learn to program computers, so we can have a better way of teaching programming.

---

[1] L/E/M: Language/Environment/Methodology

More precisely, we are concerned with how programming, using programming language such as BASIC, PASCAL, FORTRAN, or LOGO, should be taught. This is no claim that such languages are a good way to approach the task of programming, neither that learning to program is how we should approach the acquisition of computer literacy. The advocates of non-procedural language might argue that these languages will soon become obsolete, when fifth generation machines appear. However, in the next few years, we will still have to teach them, and so, we should be able to do it in the best way we can.

We examine here the cognitive psychology approach to teaching and learning, which has become in the last few years, an important field of psychology. Cognitive psychology deals with how humans acquire, organize and use knowledge:

> Cognitive psychology is the science of human information processing. Its subject matter, often called *cognition,* concerns the kind of information we have in our memories and the processes involved in acquiring, retaining, and using that information. Collectively, these processes are called *cognitive processes.* [47-p.1,2]

To study how we should teach programming, the first thing we have to know is what it is we want to teach. So we first study two approaches whose purpose is to attempt to define what kind of knowledge expert-programmers have, how it is structured and organized.

Another important prerequisite to allow for a theory of teaching is to have some idea of what it means to learn. This calls for a study of the basic principles of learning, mostly as they are defined by the branch of psychology called **Educational psychology,** always keeping in mind our goal of better teaching.

> As an applied science, educational psychology is not concerned with general laws of learning *in themselves,* but only with those properties of learning that can be related to efficacious ways of *deliberately* effecting stable cognitive changes which have social values. [2-p.8]

In the following chapters, we examine some basic approaches that might allow for better teaching, and hence learning, of programming. This involves some of the basic principles studied in the preceding chapter applied to the study of programming. This also involves studying some basic conceptual bugs (misconceptions) that novices are found to have, and trying to explain these bugs using the theory of programming knowledge introduced in chapter 2.

Finally, in the last chapter, we examine what is done at the University of Waterloo in the introductory computer science courses for non-majors, and conclude with appropriate recommendations.

2. Models of programming knowledge

To be able to teach programming, we must decide first what it is we have to teach, what it means to know how to program. Research in Artificial Intelligence (AI) [2] has shown that to become an expert in a particular domain, one has to be knowledgeable, that is, one has to possess specific knowledge about the domain.

> Expertise consists of knowledge about a particular domain, understanding of domain problems, and skill at solving some of these problems. Knowledge in any specialty is usually of two sorts: public and private. Public knowledge includes the published definitions, facts and theories of which textbooks and references in the domain of study are typically composed. But expertise usually involves more than just this public knowledge. Human experts generally possess private knowledge that has not found its way into the published literature. This private knowledge consists largely of rules of thumb that have come to be called *heuristics*. Heuristics enable the human expert to make educated guesses when necessary, to recognize promising approaches to problems, and to deal effectively with errorful or incomplete data. [17-p.4]

[2] More precisely, research in the field of expert systems, sometimes called knowledge engineering.

Programming is no exception. Expert programmer do have knowledge that enable them to write programs for solving problems, to understand programs written by other programmers, to debug or modify programs, or to learn a new programming language. In this chapter, we examine what this knowledge is, and how it is used.

In the following two sections, two models of programming knowledge are presented. The first model is based on a multi-store model of memory [3], with programming knowledge being of two different types: syntactic and semantic. The second model uses frames to represent an important part of programming knowledge: pragmatic knowledge. Finally, in the next section, we examine how those two models are related.

2.1. A multistore model with syntactic/semantic memory

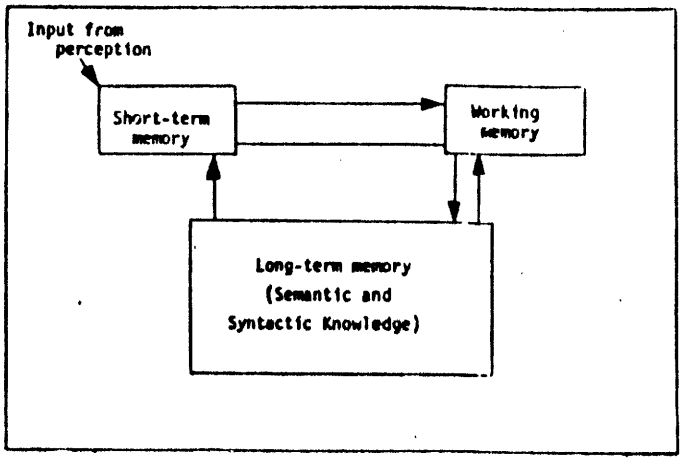
One model that has been successful in explaining many aspects of memory is the multistore model. This model views memory as consisting of many qualitatively and quantitatively different components. The two basic constituents are: the short-term memory (STM) which can only store a limited number of items for a relatively short period, and a long-term memory (LTM) whose capacity is practically unlimited, and whose content can be recalled after much longer periods of time. For an item to be stored in LTM, it has first to be received in the STM, and then be transferred to LTM. Of course, such an effective transfer may require substantial work on the part of the memorizer.

The model used by Shneiderman [36,37] is a multistore one. However, instead of having two components, it assumes the existence of three: the STM, the LTM and the working memory (WM). The STM is used to store the information received as input from the outside world. The LTM is used to store the programmer's permanent knowledge. Finally, the WM is the component where the contents of STM and LTM are integrated into new structures. These new structures may be the solution obtained to a particular problem, or it may be structures that results from

[3] The first requirement for having knowledge and being able to use it in problem-solving situations is to be able to store this knowledge. This is the reason why one of the important research areas of cognitive psychology has always been the investigation of the structure and organization of memory. The structure of memory and the structure of knowledge are like the two faces of a coin.

the understanding of a new concept, which happens in the case of learning.

The following is a schema describing the general structure of this multistore model [37-p.221]:



An important characteristic of Shneiderman's model is the way in which the content of the LTM is organized; it is divided into syntactic and semantic knowledge.

Syntactic knowledge is the type which is related to the syntax of particular programming languages. Examples of syntactic knowledge include knowing that the syntax of PASCAL while loop is: **WHILE <boolean condition> DO <statement>**, that statements are separated by ";", and knowing that "^" is used for pointers. This type of knowledge is related to particular programming languages; hence its content is compartimentalized by languages.

As most programmers know, this type of knowledge is easily forgotten. For example, if someone has been using PL/I rather than PASCAL for two years, upon returning to PASCAL, he/she will undoubtly have trouble remembering the appropriate syntax. [4]

The other constituent of programming knowledge is semantic knowledge. This represents the general concepts used in programming and which are independent of any specific language (any syntactic representation). Examples of such concepts include the notion of an assignment, a variable, an array (concept of subscripting), as well as concepts such as recursion, binary search,

[4] In a subsequent chapter, we explain why this type of knowledge is forgotten more easily. We will see that it is because it has to be learned in a rote fashion.

stack, function, procedure, parameter, etc.

An important characteristic of semantic knowledge is that it is hierarchical. An example of such a hierarchy could be:

High: quicksort.

Middle: finding largest element in an array.

Low: comparison of two values.

Semantic knowledge is more resistant to forgetting than syntactic knowledge. For example, one might forget the syntax of PASCAL's assignment statement but still remember what an assignment is.

A question we might ask ourselves is: Why this distinction between these kinds of knowledge? Many experiments have been performed which tend to justify this distinction. We now describe an interesting one.

In a first recall experiment, groups of novices and experts were given small programs and were asked to read and memorize them. Then, they were asked to write down what they recalled of the program. In that case, as we might expect, experts did much better than novices. However, in a second experiment, the programs that were given to be recalled were scrambled lines of code, that is, programs that made no sense. In this latter case, experts did as poorly as novices.

It makes sense to assume that the difference between an expert, and a novice programmer lies in the amount, and type of semantic knowledge they have. In other words, experts possess conceptual categories with which they can perceive and understand a program. Of course, this does not mean that novices lack such categories. For sure, we could expect novices to perform better than people not knowing anything about programming.

> The data suggest that both experts and novices have conceptual categories for the elements of a programming language. For the novices, the categories seem to be syntactic rather than semantic in nature and the items within the category are not related to each other in a strongly organized manner. As expertise increases, the nature of the

> categories changes. They become more conceptually complex, changing from one line operations to entire routines, and they shift from being syntactically based to being semantically based. [1-p.431]

In the first experiment, experts were able to use this semantic knowledge, which is why they performed better. But in the second experiment, this knowledge was useless, as the program was not structured correctly.

For example, in the above experiment, we can suppose that an expert, seeing some lines of code used to find the largest value of an array, will not try to remember all the details of how this is done. He/she will simply remember that the program performs such a search. [5] The reason one can do this is that he/she already has the necessary knowledge to reconstruct the details of such a search, as this is something one might have encountered often before.

Another important feature of this model is its hierarchical organization. This is an interesting feature as it confirms the approach Dijkstra conceived when talking about structured programming. An important characteristic of this approach is that of top-down stepwise refinement, which is basically a process from going to very abstract operations to more detailed and refined ones. This way of viewing the programming task is in accord with the hierarchical organization of semantic knowledge. As Dijkstra said:

> ... it might be worthwhile to point out that the purpose of abstracting is *not* to be vague, but to create a new semantic level in which one can be absolutely precise. [4-p.864]

Mayer [27,28] also points out the hierarchical organization of programming knowledge. The hierarchy he describes includes the following levels, going from lowest to highest: machine, transaction, pre-statement, statement, mandatory chunk, basic non-mandatory chunk, higher chunk, and program.

---

[5] This can be seen as a form of chunking, where input is recoded into more succinct and economic form. However, the important aspect here, is that this chunking is semantically based.

Explaining program comprehension

Using his multistore model, Shneiderman tries to explain the processes involved in different programming tasks: program composition, comprehension, modification and debugging. We only examine the task of program comprehension here.

> The first thing we might want to ask is, what does it mean to understand a program?
>
> This seems to be the best definition of program comprehension: the recognition of the overall program function, an understanding of intermediate-level processes including program organization, and comprehension of the purpose of each program statement. [36-p.31]

The reason we should view program comprehension this way is that the different levels can be understood independently. For example, suppose we have a program which performs a quicksort. One might be able to trace through the code, without understanding the overall structure of the program. One might also understand the general strategy of the program without understanding how all the low-level details fit in. Clearly, if one has to debug or modify the program, then all levels should be properly understood.

Given this definition, the task of program comprehension can then be seen as the construction, in working memory, of a multileveled internal semantics to represent the program. Using syntactic knowledge, the programmer first builds the lower semantic level. As the pieces are put together and the function of group of statements is recognized (through the use of semantic knowledge) the higher levels of the internal semantic can be built.

Is syntactic knowledge simply syntax?

Before examining the next model, there is some aspects of Shneiderman's model that need some clarifications, which he does not do. First of all, the distinction between syntactic knowledge, which is language dependent, and semantic knowledge, which is language independent, seems correct. Clearly, there are concepts which we use as programmers which are not language dependent. If this was not so, learning a second programming language would be as difficult as learning

a first one, which is obviously not the case.

Perhaps the labels "syntactic" and "semantic" do not reflect correctly this distinction. "Syntactic" knowledge, which is compartimentalized by language, should include both the syntax of the language, as given by its grammar, and its basic semantics, that is the operational semantics of its statements. [6] On the other hand, "semantic" knowledge should include all those concepts which are language-independent, but which are not necessarily semantic in the formal sense of the word, as they might include pragmatic knowledge, e.g. general problem-solving strategy.

Let's look at an example. In PASCAL's **for** loop, the value of the index variable is undefined at the exit. of the loop, which is not the case in FORTRAN do-loop. This difference, however, is not syntactic, but semantic. But both are example of loops with finite number of iterations. In other words, the syntactic knowledge should contain the fact that in PASCAL, the syntax, in its pure sense, of the for-loop, is "for <index>:=<val1> to <val2> do <statement>". But it should also contain the information that this construct is an instance of definite iteration, with the additional property that the value of <index> is undefined upon exiting.

This clarification is important as it also explains why learning a first programming language is more difficult that learning a second one. When learning a first language, the student does not yet possess the basic underlying semantic concepts, such as iteration, condition, assignment, etc., which are necessary to learn the semantics of the language. This also clarifies how the lower level of the internal semantics can be built: having recognized a syntactic construct, and having retrieved its operational semantics from "syntactic" knowledge, one then knows to which semantic concept it can be related.

---

[6] The operational semantics of a language is defined in terms of the virtual machine the language defines. It can be learned using a conceptual model of this virtual machine. The semantics of a programming language can be defined in many other ways: denotational semantics, axiomatic semantics, algebraic semantics, but these approaches are not as intuitively appealing.

2.2. Viewing expert programming knowledge as scripts

The approach we are now going to talk about is the one taken by Soloway and his colleagues [38,40,39,41]. One of their goals was to " ... attempt to identify the needs of novice programmers by understanding the source of their difficulties." To do so, they focus their attention on programming: "what difficulties do non-professional programmers have in learning to program, and what are the source of these difficulties." [40-p.28]

Their first step was to study the bugs and misconceptions that novice programmers do and have, and then to try to explain them based on the underlying assumption that these misconceptions are not random:

> The key to understanding the misconceptions of non-professionals is a theory of programming knowledge, i.e., the knowledge which expert programmers have about programming. We assume that misconceptions are some variant of such expert knowledge. [40-p.28]

We now examine the basic ideas of this theory of programming knowledge. Then we briefly look at the bugs and misconceptions that were found and see how they can be explained, how they relate to the theory.

An important basis of this approach is the following idea:

> We believe that experts use more than just knowledge of the syntax and semantics of programming language when they write programs to solve problems. Expert programmers have and use high-level, plan knowledge to direct their programming activities. [40-p.30]

Another type of knowledge that experts have is the one concerned with the rules of programming discourse, which are "rules that specify the conventions in programming". [43-p. 154]

Programming plans and variable plans

The most important part of an expert-programmer knowledge is, according to Soloway, his/her knowledge of programming plans.

> In our view, programs are composed of programming plans that have been modified to fit the needs of the specific problem. The composition of those plans are governed by rules of programming discourse. [43-p.154]

A programming plan is a structure that helps a programmer to create a program given a particular problem, that "help him/her relate problems to programs." [39-p.208]:

> A plan is a procedure or strategy in which the key elements of the process have been abstracted and represented explicitly. Faced with a new problem, an expert retrieves plans from his/her knowledge base which have proven useful in similar situations, and then weaves them together to fit the demands of the new problem. [40-p.30]

Another way to view the idea of programming plans is to look at them as programming scripts. A script is a stereotypic action sequence. An example of such a script is what Soloway calls the TOTAL_RUNNING LOOP PLAN. For example, consider the problem of determining the average of several numbers. Basically, this requires using three variables (New_value, Running_total and Counter), initializing them, building the running total in the loop and incrementing the count, etc. Depending on the condition terminating the loop, different specialization of the basic scripts can be used, but the basic elements remain the same.

The existence of scripts was proposed as a result of the following experiment. The participants were given small programs with lines left out. They were then asked to fill in the missing lines. It was found that experts performed better than novices, which was explained by the fact that they could recognize which script was being used. [7]

---

[7] This is another important aspect of scripts; as they represent stereotypic situations, a lot of information need not be told explicitly. For example, if someone tells us that he went to a restaurant, then we can assume without much risk of error, that he looked at the menu, ordered some food, ate it, paid the bill, and left. The same is true with programming scripts.

The programming plans they identified are of two types: looping plans and variable plans. To represent the type of knowledge they encountered, they decided to use frames. [8] A schema of the type of structure obtained is given in Appendix 1.

As can be seen, the organization of plan knowledge is, once more, hierarchical. The first level is that of strategic plans. These kind of plans define the global strategy used in an algorithm. In the case of looping plans, there are two basic such categories: the read/process looping strategy and the process/read looping strategy.

The next level consists of tactical plans, which specify a local strategy for solving a problem. For example, to solve the problem of computing the average of ten numbers, one may decide to use the counter_controlled running_total looping plan, which is a tactical approach for a read/process looping strategy.

Finally, there are implementation plans. These are language-dependent constructs to realize the tactical and strategic plans. For example, a counter_controlled running_total loop for solving the problem of computing the average of ten numbers might be implemented in PASCAL using a **for-** loop. [9]

There is another form of knowledge which programming experts have, but which novices may lack. This other aspect, which has also been studied by Soloway and his colleagues, concerns programming plans related to the role of variables:

> In addition to identifying abstract programming plans, we sought to understand the role variables play in programs. Just as actors take on different roles in a play, variables take on different functions in a program. ... That is, a variable can be perceived as simply containing an arbitrary value, or it can be perceived as containing values which have *a specific purpose.* The roles that we have described reflect the special purpose which values, and their variables, play in programs [40-p.34,italics are mine]

---

[8] Note that they never say that knowledge is indeed represented in the expert mind as frames; all they say is: " In order to be precise about the plan knowledge which expert programmers have, we decided to encode this knowledge in a "formal language". That is, we have borrowed a language for representing knowledge from Artificial Intelligence called "frames" ". [40-p.31]

[9] Of course, it could also be done using a **while** or a **repeat**, but in this case, the **for-** loop is the most appropriate construct.

On a basic semantic level, a variable is a location in memory where information can be stored and retrieved, this information being either a simple value or a complex structure. But on a pragmatic level, a variable used as a counter and one used to accumulate some total are very different. The syntax and semantics of a language is not sufficient to describe those differences.

Rules of programming discourse

Another important aspect of an expert programmer's knowledge is concerned with the rules of programming discourse. These rules specify how the basic elements of a program should be put together, in ways which are "... analogous to discourse rules in conversation" [43-p.154].

> In our view, programs are composed from programming plans that have been modified to fit the needs of the specific problem. The composition of those plans are governed by rules of programming discourse. [43-p.154]

There are many examples of such rules: a variable name should agree with its function; a variable should not do double duty; code which is never used should not be included; etc. [10]

Following the rules of programming discourse is very important as "these rules setup expectations in the minds of the programmers about what should be in the program." [43-p.154] In an experiment described by Soloway [44], it was shown that the performance of experts in understanding programs which violated one rule of programming discourse decreased considerably, sometimes becoming worse than that of novices.

Explaining program comprehension

Assuming that we indeed represent programming knowledge using schemata structures, we can use the following view of frames as **active data structure** to explain the comprehension of a program:

---

[10] Kernighan and Plaugher, in their book: *The Elements of Programming Style* [21], describe many of those rules of programming discourse.

> Schematas activate themselves whenever they are appropriate to an ongoing analysis, and they are capable of guiding the organization of the data according to their structures. [32-p.44]

When a program is read, its basic syntactic constructs and elements are recognized. By seeing the way in which these components are used and interrelated, the appropriate frame structure can be recalled/used to stand for them. Of course, this assumes that they are used correctly, that they do not contradict the programmer's expectations. If this is not the case, a different understanding process must take place, which requires a deeper understanding of the program, e.g. by simulating the program in a bottom-up fashion. [11]

2.3. The relation between Shneiderman's and Soloway's view

In many ways, Shneiderman's point of view is similar to that of Soloway's. Both agree on the fact that programming knowledge is more than simply knowledge about the syntax and semantics of programming language. For Shneiderman's, this basic knowledge is represented by the syntactic knowledge contained in LTM, whereas the semantic knowledge contains knowledge about language-independent concepts.

Soloway's work can be seen as an elaboration of the structure of semantic knowledge possessed by programmers. All that Shneiderman does is give some of the elements contained in it, and stress its hierarchical organization. However, he does not describe how it is organized.

> Shneiderman (1976) and Adelson (1981) have shown that experts seem to use high-level knowledge to understand programs, while novices tend to focus on the specific statements employed in a program. Building on this work, it is our goal to identify *specific* knowledge difference between experts and non-experts, and examine how different levels of knowledge affect the performance of non-experts. [40-p.30]

---

[11] Soloway [44] describes these two kinds of understanding as *shallow* understanding process, and *deep* understanding process.

2.4. Models and "Reality"

As more work is done in this area, a better understanding of the organization of programming knowledge, of the relation between the various levels will be obtained. As Soloway says:

> We do not mean to suggest that the knowledge incorporated into figure 1 [12] is the final word. .... Regardless of the details, experts do have and use structures similar to those in figure 1, and we should be teaching students based on this view. [39-p.210]

A philosophical question we might ask ourselves is: Do we really have such structures in our mind? Do they constitute an "objective" reality in the mind of programmers? Or are they simply, as Greeno says, talking about problem-solving, a framework for our own understanding:

> These are not theories in the sense of generating empirically falsifiable hypotheses about performance in any specific experimental situation. Rather, the proposal I have presented here, like the others in the field of problem-solving, constitutes *a conceptual framework that may be useful in enriching our understanding of processes that take place in problem-solving.* [12-p.131,italics are mine]

In this regard, a great deal could be learned from modern physics:

> The crucial feature of atomic physics is that the human observer is not only necessary to observe the properties of an object, but is necessary even to define these properties. In atomic physics, we cannot talk about the properties of an object as such. They are only meaningful in the context of the object's interaction with the observer. In the words of Heisenberg, "What we observe is not nature itself, but nature exposed to our method of questioning". [3-p.152]
>
> A careful analysis of the process of observation in atomic physics has shown that the subatomic particles have no meaning as isolated entities, but can only be understood as interconnections between the preparation of an experiment and the subsequent measurement. [3-p.78]

[12] See Appendix 1.

The same holds when psychological experiments are performed and cognitive structures are inferred from them:

> Since we cannot look into people's minds and observe network diagrams, knowledge structures must be inferred from their behaviors. [31-p.206]

Even though mathematics is used in quantum mechanics to describe the properties and relationships of atomic particles, nobody would believe that a particle is the same as the equation describing it. The same should be kept in mind when working with descriptions and representations of knowledge: the map is not the territory.

> Because our representation of reality is so much easier to grasp than reality itself, we tend to confuse the two and to take our concepts and symbols for reality. [3-p.35]

3. Learning

In this chapter, we briefly examine some basic principles of learning. That is, we try to see what it means to learn something. We first examine some general definitions of learning as given by different branches of psychology. We then examine two different views of learning, that of Ausubel and that of Rumelhart. Finally, we look at the concept of meaningful learning, which is the type of learning necessary in teaching any meaningful domain.

3.1. Some general definitions of learning

For many years, psychologists studied learning in their laboratories. As the dominant paradigm at the time was that of behavioural psychology, many of the definitions of learning are behaviour-related:

> [Learning is] a relatively permanent change in behavior potentiality that occurs as a result of reinforced practice. [18-p.5]

> [Learning is] any process that modifies a system so as to improve, more or less irreversibly, its subsequent performance on the same task or tasks drawn from the same population. [24-p.367]

As cognitive psychology gained more importance, learning started to be viewed as change in the learner's cognitive structures. That is, the change in behaviour is not negated. However, those changes are present only because the learner's knowledge structure has been modified. These modifications of the cognitive structures should also be taken into consideration when teaching. Teaching should then have two kinds of objectives: [13]

[1] Certain behavioural objectives, e.g. to acquire problem-solving ability in a particular domain of expertise.

[2] Certain cognitive objectives, e.g. to acquire the knowledge structures of experts.

In the next section, we examine in more detail the different ways in which learning, as change in cognitive structures, can occur. In general, this is the kind of learning with which we are concerned. As our goal is to teach programming, of course we are concerned with improving the problem-solving ability of students as it involves the use of computers. However, as we soon see, a necessary prerequisite for problem-solving ability is the existence of an adequate background of knowledge, and it is the acquisition of this knowledge that will be our main concern.

3.2. Different processes of learning

We now examine the kind of learning which can be considered when we talk of learning as resulting in changes in the cognitive structures of the learner. We examine two approaches and then try to see how they relate to each other.

Meaningful vs rote learning

Ausubel is an educational psychologist who, in his book: **Educational psychology - A cognitive view** [2], tries to describe the general principles of learning as they relate to classroom learning. His goal is to examine these principles so that they can be influenced to facilitate learning of students. The important dimension we now examine is the distinction between meaningful learning and rote learning.

---

[13] For more details, see [13].

For Ausubel, classroom learning involves the acquisition and retention of large bodies of knowledge. These bodies of knowledge are essentially bodies of meanings, that is, to acquire knowledge is to acquire some new meanings. The acquisition of new meanings is the essence of meaningful learning.

What are meanings?

> [Meanings] refers to the differentiated cognitive content evoked in a given learner by a particular symbol or group of symbols, after either of these expressions has been meaningfully learned. [2-p.42]

So meanings are simply seen as the resulting knowledge structure that exist in the learner's mind after some meaningful learning has occurred. Meanings, then, are simply the outcome resulting from a meaningful learning process.

Meaningful learning is a process which results in the emergence of new meanings in the learner's cognitive structures, and hence, in changes in this structure. One of its most important characteristic, its **essence,** as Ausubel says is that "... symbolically expressed ideas are related in a non-arbitrary, substantive fashion to what the learner already knows [2-p.37]".

Substantive relatability is when the meanings to be acquired do not depend "on the exclusive use of signs and symbols" , i.e., what has to learned need not be memorized in a verbatim fashion, in an exact word-by-word basis. Non-arbitrary relatability means that there exist an "adequate, self-evident basis" to which the new meanings can be related. Example of such basis are generalization, specialization, extension, elaboration, derivatives, qualifications, analogy, etc...

Another psychologist has a similar view:

> ... learning is a function of the abstract and distinctive, concrete associations which the learner generates between his prior experience, as it is stored in long-term memory, and the stimuli. Learning with understanding, which is defined by long-term memory plus transfer to conceptually related problems is a process of generating semantic and distinctive idiosyncratic associations between stimuli and stored information. [48-p.89]

What happens, then, when the new meanings to be acquired are related in a verbatim (non-substantive) and arbitrary fashion? In this case, we are dealing with rote learning. An example will help clarify the distinction. Suppose we are trying to learn the following list of pairs:

awej - eeg

sdf - ddl

ldk - wwfd

losm - egdq

The first thing to note is that, here, the learning is verbatim by nature. The words of each pairs have to be recalled exactly as they are. Also, the elements of the pairs can be related to each other only in a purely arbitrary fashion. And the task in general is also totally arbitrary, as it cannot relate in any way to what the learner already knows. This is an example of pure rote learning.

So what is involved in the process of meaningful learning? First, some potentially meaningful material has to be presented to the learner. Then, this material has to be made meaningful in a process of internalization which involves grasping the substantive content of the material (extract its essential idea/concept) and relating this content, in a non-arbitrary way to what the learner already knows.

Meaningful learning is only possible if the two above conditions are present; the material must be logically meaningful **and** the learner must exhibit a meaningful learning set. This meaningful learning set is an important part of the process of meaningful learning as it makes it an active process rather than a passive one. If the material being presented to the student is logically meaningful [14] but the student does not try to relate it to what he/she already knows by analyzing it, decomposing it, examining it, etc., then it will not be learned meaningfully.

In a subsequent section, we examine why these characteristics of meaningful learning (substantive and non-arbitrary relatability) are so important. Let us simply remark that the distinction we introduced earlier between syntactic and semantic knowledge can be related to these ways

[14] That is, it could be meaningfully learned by some other person having the same background knowledge.

of learning. Syntactic knowledge is essentially acquired through rote learning. The details of the syntax of a language have to be learned precisely without any modifications. The fact that PASCAL's comments have to be enclosed between (* *), whereas in ADA they are denoted by -- is totally arbitrary. On the other hand, semantic knowledge is learned through meaningful learning. Learning how a bubblesort works is not done in a verbatim manner. The idea of bubblesort can be grasped without references to the details of any particular programming language, and its underlying idea can be grasped non-arbitrarily, by using an analogy with bubbles going up.

Accretion, tuning and restructuring

This view of learning is interesting as it assumes that knowledge is represented using frames, much in the same way that expert-knowledge was represented in Soloway's view. The main goal of learning is seen as the formation of new cognitive structures as opposed to the simple accumulation of knowledge.

There are three basic modes of learning: accretion, tuning and restructuring.

Accretion

Accretion is the accumulation of factual information. It is present when learning relatively simple information for which there already exist concepts. The existing frame structures are used to generate new data structures by a simple copying process. This is the kind of learning involved in the acquisition of the memories of a day's event, when learning names, phone numbers, etc.

Tuning

This mode of learning involves some modification to the existing frame structures. However, these modifications are minimal, as they don't change the basic structure and relationships.

> [The] basic relational structure of the schema remains unchanged, and only the constant and variable terms referred to by the schema are modified. [32-p.47]

Examples of such modifications are improving the accuracy of elements, generalizing, specializing, or representing constraints on the use of a frame so it will not be invoked inappropriately. [15]

[15] As Howe points out, a frame "also contains information about how the frame should be used, information about what can be expected to happen next and information about what to do if these expectations are not ful-

Restructuring

This mode of learning is the most important one as it is required for more complex learning tasks. In this case, there may not exist structures to which the new elements can be fitted. A reorganization of some part of the memory may be required. In other words, it involves the construction of new frame structures when the existing ones cannot be used adequately.

> Indeed, often the point of learning is the formation of the new structures, not the accumulation of knowledge. Once the appropriate structures exist, the learner can be said to "understand" the material, and that is often a satisfactory end point of the learning process. [32-p.39]

Of course, a first phase of accretion of new information may have preceded this restructuring:

> In part, it is the unwieldiness and ill-formedness of this accumulated knowledge that gives rise to the need for restructuring. [32-p.39]

An important question is how does learning by restructuring occur, that is, how are new frame structures created/generated? It seems that analogy is an important process in this mode of learning:

> We believe that the typical course of such a learning process consists of an initial creation of a new schema by modeling it on an existing schema. This new schema, however, is not perfect. It may occasionally mispredict events and otherwise be inadequate. We then believe that the newly acquired schema undergoes a process of refinement that we have dubbed *tuning*. [33-p.357]

We then have a process of learning which looks something like the following: New information is encountered. An attempt is made to understand it in terms of the existing knowledge structure. If this is not possible, in the case where no schemata/frames can account completely for it, then the existing frames that can account for the essential features of the situation may be used to generate a new schemata. This new schema differs from the old ones in the way the existing

---

filled." [19-p.261]

structures did not account perfectly for the new information.

Features of the old schemata that were not contradicted by the new situation are also carried over. In this way, a substantial amount of learning can occur as these features are used to infer properties of the new situations. Of course, this may cause problems, as those properties may not be valid. These new structures then have to be tuned to account for these discrepancies.

The three modes of learning and learning LOGO

Howe [19] uses this view of learning to explain how young children learn to program in LOGO. Children seem to go through three phases: product-oriented, style conscious and creative problem-solving.

In the first phase, the children's desire is to "produce particular drawings, of effects, without any real concern for the way in which these effects are achieved." [19-p.256] In this stage, the three modes of learning are present:

> They learned by accretion new terminology such as the names of the items of equipment, the names of commands, and so on. They tuned memory structures representing concepts like locomoting about in space to handle the drawing tasks used in the early days of learning to program. They began to build new memory structures to represent these skills which they had to acquire to solve the problem of getting information into and out of the machine. [19-p.261,262]

In the style conscious phase, "a pupil's main concern is to work in what he perceives as correct LOGO programming style" [19-p.257], even though this may sometimes be done inappropriately. [16] This is done "through the construction of new, but as yet unrefined, frame structures".

> The essential point is that these structures are used indiscriminately to begin with, producing the inappropriate but correct behaviour which we observed. However, they are generally refined into new, more specific frame structures which carry more precise in-

[16] For example, the pupil may "slavishly break a problem into sub-problems, or introduce redundant inputs into procedures, or introduce redundant commands" [19-p.258].

> formation about the situation in which they should be applied, reducing the possibility that any one will be used inappropriately. [19-p.262]

In other words, the initial phase leads the student into the generation of new frame structures concerned with good programming style: sub-procedures, parameter passing, recursion, mnemonic names, etc. However, the student may not know exactly yet, in what context these concepts have to be used. By using them, playing with them, their correct use can be understood.

Finally, the last phase is concerned with creative problem-solving using the concepts that have been acquired. In the previous phases, this type of problem-solving could not be present as "a pupil cannot work creatively until he has built new mental structures to represent the essential concepts" [19-p.262]. It is the style conscious phase which allows these basic concepts to be represented in well tuned structures and to be made available for such creative problem-solving.

How do those two models relate?

The following describes my own understanding of the relation between the two models of learning we just examined. The first relation that seems to be clear is the correspondence between rote learning, and accretion learning. In rote learning, we saw that the relatability of the new material to existing knowledge was arbitrary and verbatim. Of course, if we think in terms of frame structures, some existing frame structures have to be used to represent the newly learned material. However, because this learning is done in a rote manner, these frame structures do not correspond to high level concepts. Rather, they must be of a relatively low-level. For example, when learning a list of pair-associations, we might do this using frame structures describing the concept of pair-association, with the slots being filled with the words to be related. [17] This can be seen as the accretion of new information using this "low-level" frame structure.

In the case of meaningful learning, the substantive grasp of the content means that appropriate existing frame structures are retrieved from memory. The fact that they are related non-arbitrarily to existing knowledge may mean that a new frame is created whose slots are pointers

---

[17] If the paired elements did not have to be learned in a verbatim fashion, the slots could be filled with pointers to frame structures representing the substance/concept of the elements.

to the frames associated with the substantive content to be learned. The non-arbitrary relatability is then represented by this newly created frame structure. Again, generating this substantive, non-arbitrary relatability requires an active involvement from the part of the learner. He/she has to recognize that none of the existing structures can account perfectly for what he/she is trying to learn. Once this is recognized, he/she must seek ways in which they can be related to each other, so appropriate frame structures can be created.

Based on these two models, one might think that the accretion/tuning/restructuring model is in the same relative position to meaningful learning as Soloway's view is to Shneiderman's, that is, as an elaboration of the latter. Rumelhart's model, by using an explicit knowledge representation, clarifies what is meant by substantive and non-arbitrary relatability; new frame structures are created.

3.3. Meaningful learning

In this section, we examine in more detail the principles of meaningful learning to see what are the basic prerequisites for such learning to occur. We saw that meaningful learning is characterized by non-verbatim, non-arbitrary relatability of the new material to the existing knowledge already present in the learner's cognitive structures. Why is this so important? What is the impact of such a type of learning? In what ways is it better than rote learning? What are the cognitive variables that can be influenced so that meaningful learning occurs? Those are all questions we now examine.

The "advantages" of meaningful learning

The first important aspect of meaningful learning which stems from the non-arbitrarily relatability is the better retainability of learned material. Material learned meaningfully is retained/remembered for longer periods of time than material learned in a rote fashion. Material which is rotely learned is forgotten more easily because, being related arbitrarily and non-substantively, it may suffer from interference with newly learned material. That is, rote learning of new material may interfere with material already learned in a rote fashion because it requires

the **addition** (or accretion) of new information rather than their **integration.** For example, learning the syntax and semantics of PASCAL can be done quite easily if the basic concepts of programming are already known. However, if at the same time, or a little later, another language, let's say ADA, is learned, then PASCAL's syntax will be remembered with more difficulty, as both languages have a similar syntax, but yet, are not exactly the same. [18]

Another advantage is that material which has been learned meaningfully, and then forgotten, is relearned much more easily than if it had initially been acquired rotely. The initial substantive, non-arbitrary relatability that was generated, but then forgotten, can be generated more easily as it was done before. [19] If the material was initially learned rotely, then meaningful relearning cannot be helped by the original substantive, non-arbitrary associations/relations, as they were never generated.

Of course, the most important advantage of meaningful learning is its influence on problem-solving ability. Meaningfully learned material can be transferred to help in problem-solving. [20] On the other hand, rotely learned material, which only consists of verbatim propositions, cannot be of any use, unless the problem to be solved is identical to one that has been encountered before, that is, requires no transfer.

Meaningful learning and rote learning are related to each other in the same way that understanding a proposition, and memorizing it on a word by word basis are related. To be able to solve some new problems in a way which is not a simple trial-and-error process, one has to understand what one is doing. For example, if one knows what an integral represents, one will be able, given a picture and the equation of a 3-dimensional figure, to find its volume. On the other hand, if one only learned the formulas to solve particular equations, one might not be able to determine which formula to use, or to determine what the bounds of the integral should be.

---

[18] In this case, it is the similarities between both languages that may lead to confusion. Having just learned that ADA IF-statement must be terminated by ENDIF, and knowing that PASCAL also have IF-statement, one might infer that they must also be terminated with ENDIF.

[19] In terms of frames structures, this means it is easier to generate those new structures if they have been encountered before. Although they may not be present in memory anymore, part of the process that lead to their initial generation may still be remembered.

[20] By transfer, it is meant the possibility of using knowledge in new situations.

The necessary conditions for meaningful learning to occur

For meaningful learning to occur, the new material must be related substantively and non-arbitrarily to what the learner already knows. Clearly, then, the first condition for such learning to occur is that the material being presented be logically meaningful. [21]

Once the material is presented, it has to be meaningfully acquired by the learner. Of course, this is only possible if the learner has the necessary prerequisite or background knowledge to which the new material can be related. If this background is absent, the material can only be learned rotely.

The crucial role of an appropriate background can be seen when learning a programming language. Learning the first language can be difficult because the basic semantic concepts (variables, assignments, iterations, procedures, etc.) first have to be learned. Once acquired, these concepts can be used as background knowledge that will facilitate the learning of subsequent language.

Another condition that must be present for meaningful learning to occur is that the learner exhibit a meaningful learning set. Even though the material being presented is logically meaningful, and the learner does possess the necessary background knowledge, if he/she does not try to extract the substance of what is being presented or does not try to generate associations between what is being presented and what he/she already knows, meaningful learning will not occur.

Mayer [25,28] also describes the learning process in much the same way. Learning is seen as a process requiring 3 conditions

1- Reception of information

2- Presence of prerequisite knowledge

3- Activation of an assimilative set

---

[21] We are concerned only with reception learning, as opposed to discovery learning. In discovery learning, the meaning to be acquired has first to be discovered through an initial phase of problem-solving activity. Reception learning is the most important mechanism used in classroom learning.

Whenever one of these conditions is missing, meaningful learning cannot occur. Examples of situations where one of the condition is not present are the following:

[1] The student may not effectively determine what has to be discovered, which can happen in discovery learning.

[2] The prerequisite knowledge is not present, or not recalled because its pertinence is not recognized.

[3] The learner is not intellectually involved. He/she may answer general questions but cannot generalize from one problem to another.

3.4. The role of practice and problem-solving

Practice and problem-solving can play an important role in the meaningful learning process.

> [Practice] influences cognitive structures in at least four different ways: it increases the dissociability strength of the newly-learned meanings for a given trial and thereby facilitates their retention; it enhances the learner's responsiveness to subsequent presentations of the same material; it enables the learner to profit from inter-trial forgetting; and it facilitates the learning and retention of related new learning tasks. [2-p.274]

Even though the presence of knowledge is necessary for genuine problem-solving, it is not sufficient. The relevance of the particular knowledge has first to be recognized. This can be done by having students solve problems which require using the particular knowledge they just used, making explicit the conditions under which it has to be used.

Again, practice and elaboration activities are very important for meaningful learning because they encourage the activation of the existing knowledge that is relevant for comprehending the newly presented material. In other words, problem-solving and elaboration encourage the learner to have a meaningful learning set.

An important difference that we must keep in mind when talking about problem-solving is the one between meaningful problem-solving and solving by trial-and-error. For instance, in computer programming, it is often very easy to generate a program that will partly solve the problem,

without having a clear understanding of the solution, without really understanding how the program works when it is finished. [22] Too often, programs are obtained, not through a real design process requiring a genuine understanding of the problem to be solved, but through many successives stages of coding, debugging and patching. The final result may have a behaviour which is almost correct, with many bugs being present. However, it has no meaningful structure, as the student did not really understand what he/she was doing.

4. Teaching/learning to program

This chapter examines the problem of teaching programming to novices, that is, individuals who have never programmed before. In addressing this problem, it is important to recall both the ideas of what constitutes programming knowledge, and the general principles of learning that were introduced earlier. We first examine some of the problems that are encountered when teaching programming. Then, we look at teaching strategies that might help alleviate these problems.

4.1. Some basic problems of learning programming

The need for background/context to anchor new ideas

As we saw earlier, meaningful learning requires that new ideas be related to existing knowledge. However, one important problem with learning a new domain is that such anchoring/background knowledge may not be available.

> The lack of adequate context for the beginning programmer stems from the fact that a computer is unlike anything which the student has previously encountered, nor is programming very much like any problem-solving behavior in which the student has previously engaged. [22-p.308]

[22] As Polak [30-p.5] points out, talking about program verification: "People unfamiliar with verification do not realize the importance of theory in writing programs. Programming is misleading in this respect because it allows us to write programs without having completely understood the problem. Verification makes us aware of all the mathematical properties (or alleged properties) we use in a program."

If we want students to learn a programming language, they will have to learn both its syntax and its semantic. As we explained earlier, learning the syntax of a programming language is done mostly by rote. However, is it the same with the low-level semantics, that is, the semantics of its statements in terms of the modification that they generate in the computer? Is there a way in which the basic semantics of a programming language can be learned meaningfully? Can we provide students with some knowledge that will be available for anchoring the concepts related to the semantics of a programming language?

The answers to the above questions are affirmative. The desired result can be obtained by introducing explicitly to students a conceptual model of the computer. Such a model contains the basic elements necessary to relate the instructions of the programming language to the corresponding changes occurring inside the machine. When the model is understood, the semantics can then be learned meaningfully more easily, as its description can be related to the modifications it generates in the model of the machine. We examine in more detail the use of conceptual models in the next section.

Another way to ensure that students always have the prerequisite background knowledge is to organize the material to be taught in such a way that prerequisite knowledge is always introduced explicitly before it is needed. An approach that can help ensure this is the spiral approach, which we examine later.

The lack of vertical transfer

Another problem which can occur is that students may have trouble effectively using the knowledge they just acquired. They cannot activate the process of vertical transfer.

> Transfer is the facilitating effect of previous learning on new learning.
>
> ... Vertical transfer occurs when rules or concepts learned at a "lower" cognitive level (such as *concept learning*) are applied at a "higher" cognitive level (such as *problem solving*). [22-p.308,309]

For example, one may know how a while loop works and understand its syntax and semantics, but

still have difficulty in employing it.

An effective way in which the student can be helped to generate the desired associations/relations is by having appropriate exercises that will clarify the context in which the newly acquired concepts can be used.

The explicit introduction of programming scripts is another way which may help bridge the gap between knowing and doing, which may allow for better vertical transfer. Kreitzberg [22] describes this aspect in terms of meta-rules applied by the programmer.

> Thus a proficient programmer, working on a problem that is within his experience, is hypothesized to operate primarily at a synthetic level. That is, he has learned or discovered the techniques or "meta-rules" that might be applicable to his specific problem, and applies these techniques in a relatively straightforward manner. [22-p.309]

These meta-rules are nothing but the programming plans and scripts of Soloway.

Computer shock

A problem that may occur when learning a new subject is what is called an *initial learning shock,* or in the case of programming, *computer shock.* This computer shock is the result of the misconceptions students might have about computers, misconceptions which lead them to be uncertain about what they understand or not. It is also plays an important role in the fact that the learner does not exhibit a meaningful learning set, because he/she may be cognitively overwhelmed by all he/she has to learn: syntax and semantics of the language, basic programming concepts, text editing, program compiling, files manipulation, etc.

> Faced with uncertainty and accompanying panic, the student tries to learn by rote lest he lose grasp of the material completely. The result is that synthesis of ideas is difficult if not impossible. [22-p.310]

This uncertainty is mainly caused by the unfamiliarity that occurs when students are first confronted with computers, and "the necessity to operate with a high degree of precision and at a level of detail which seems demanding to the student" [22-p.310]. As any advanced programmer

knows, computers can sometimes be very stupid, requiring an incredible amount of precision, even on non-meaningful details. This, at first, seems very hard to understand for the novice, who has heard of computers as intelligent machine, as *cerveau electronique.* To discover that computers are unable to recognize simple typing mistakes, or to be told that the program cannot work because a semi-colon is missing at line 42 is frustrating. If the computer can know that there is a semi-colon missing, why can't it add it itself?

The utilization of a conceptual model can also help solve this problem. This model, which can be used to reason about what happens inside the computer, makes the computer understandable, rather than something mysterious and unpredictable.

Conceptual bugs

There are many misconceptions that novice programmers may have and which may lead them to write erroneous programs. Soloway and colleagues [39,40] performed many experiments with novice and intermediate programmers to determine the kind of misconceptions they had about programming concepts. They asked students to solve three different problems, all involving the computation of the average of numbers. The difference between the three consisted mainly in the way the test to terminate the loop had to be constructed, so that the best solution involved using one of the three looping construct *(while, for* and *repeat).*

The first aspect which they studied concerned how well students could distinguish the three looping constructs of PASCAL. They found that most students could not choose the appropriate construct and use it correctly [39-p.212]. For example, many students did not use the *for* loop appropriately.

> ... since the *for* loop does so much work automatically, we thought it would be the easiest to understand and use. On second thought, we decided that these automatic, implicit aspects of the *for* loop might in fact be the problem. Since the *for* loop does a number of actions automatically, students might be uncertain about exactly how it works. [40-p.41,42]

The same kind of problem was found with both the *while* and the *repeat* loops, in that students did not know in which context they should be used. Soloway and his colleagues claim that this confusion results from the way they are usually distinguished:

> As opposed to these "syntax level" descriptions, we feel that the "deep structure" of these constructions should be emphasized. A better explanation is: If the test variable will have a meaningful value as the loop is entered, i.e., a value that could prevent the loop from being executed even once, then a *while* loop is appropriate. If, however, the first meaningful value of the test variable is assigned to it during the loop, then a *repeat* loop is the appropriate construct? [40-p.43]

In the same experiment, they also found that, even when the appropriate looping construct was chosen, this did not imply that the program would be correct.

> The students (novices and intermediates) who chose the appropriate loop construct were only slightly more likely to write correct programs. In other words, simply choosing the appropriate loop *construct* is not a predictor of program correctness. ... [But], choosing the appropriate looping *strategy* can be a predictor of correctness. [40-p.43]

Two basic looping strategies were identified, which they called the read-i/process-i and process-i/read-next-i strategy [See Appendix 1]. They found that students prefer to use the first strategy, and when they could use it, they tended to have fewer bugs [42]. However, a *while* loop is better suited to the second type of looping strategy, and it is this construct which is used by students most of the time. As a student told them: "When I don't know what is going on, I use a *while* loop." [40-p.41].

Other conceptual bugs regarding the *while* statement were discovered. For example, many students seemed to think that, whenever the condition became false inside the body of the loop, the loop was exited. This misconception was explained as an overgeneralization of the semantic of a while statement in natural language [39-p.217]. [23]

---

[23] This is an example of the way analogies might sometime be harmful. Some undesired properties are carried over through the analogy. For more details, see [15]

Differences in the way variables were treated were also discovered. The variable involved in accumulating the total caused more bugs than the one used as a counter. On a syntactic/semantic level, there is no difference between the two, as both are variables, modified using assignments. For example, in updating the running total variable (Z:=Z+X), many students split the instruction into two (Y:=X+Z;Z:=Y), which they never did for a counter variable. It seems students saw the instruction "I:=I+1" as a special kind of entity, rather than a particular type of assignment.

Another misconception involves read variables, i.e., variables whose new values are obtained with *read* statements. Students did not understand that a *read* statement, in addition to reading a new value, assigns this value to a variable. Also, some students, to keep track of the previous value, tried to obtain it by substracting one from the current value, as would be done with a counter variable. They explain this misconception in the following way:

> Since these variables are "sibling" concepts [See Appendix 1], this confusion seems quite plausible. Thus, a student might well believe that if substracting 1 from the value of a Counter Variable will return the previous value of the Counter, then substracting 1 from the variable Num should also return the previous value of Num. [40-p.50]

Finally, there was also the problem of "mushed variables":

> While experts can distinguish among different kinds of variables, and correctly note when such variables should be used, we found that a substantial percentage of novice programs (27%) used the same variable incorrectly for more than one role. [40-p.51]

For example, some students used the same variable to keep track of the running total and to read the new value.

4.2. Strategies for teaching programming

4.2.1. What should be taught in an introductory course?

> To the teacher of programming, even more, I say: Identify the paradigms *you* use, as fully as you can, then teach them explicitly. They will serve your student when FORTRAN has replaced Latin and Sanskrit as the archetypal dead language. [10-p.459]

A great deal has been written on what should be taught in an introductory programming course [11,14,16,20,29,34,35,49]. Most of the discussion is centered around whether problem-solving techniques should be taught explicitly and around the idea that programming is more than coding. As discussed earlier, programming knowledge should be seen as consisting of syntactic knowledge (programming languages), semantic knowledge (programming concepts), and pragmatic knowledge (programming scripts and plans). All of these aspects should be taught. Computer programming is neither coding nor problem-solving per se. It is problem-solving using computers.

It is here that Soloway's work becomes particularly important. The kind of knowledge he describes can be seen as a bridge over the gap existing between the knowledge of particular programming languages, and the knowledge about general principles and heuristics for problem-solving. The pragmatic knowledge he describes, in particular programming scripts, is the kind of knowledge that makes it possible to effectively solve a problem on a computer using some programming language.

For example, given a problem and some general problem-solving strategy, e.g. divide-and-conquer, we can use such pragmatic knowledge to generate a solution that will be in the realm of computer implementability. In other words, it is not enough to know that a problem can be solved by decomposing it into subproblems, and solving the subproblems. It must also be known what kind of subproblems can be reached from a problem, what kind of subproblems can be/should be attained. Programming plans and scripts represent such entities into which problems can be decomposed.

To solve the problems that will be given to them, students will have to learn those scripts, to acquire the paradigms required to solve them. As Kuhn points out

> After he [the student] has completed a certain number [of exemplar problems], which may vary from one individual to the next, he views the situation that confronts him as a scientist in the same gestalt as other members of his specialists' group. [23-p.189]

In other words, to solve correctly the given programming problems, the student will have to learn and use the scripts implicitly required by the problems. The trouble is that he/she may not discover the necessary scripts, and so will be unable to solve the given problem.

Students, then, should be introduced explicitly to the basic programming scripts they will have to use, including the basic looping strategies and the different ways and contexts to use them, the different roles which variables can play. And the same should be done with the basic rules of programming discourse.

> If such meta-rules can be identified and taught to the student he should be able to develop programs far more rapidly and with greater ease, since he no longer needs to invent these constructs each time he needs them. [22-p.311]

4.2.2. Using conceptual models

The characteristics of a conceptual model

DuBoulay et al. [6] describe the two basic characteristics of a conceptual model, which are simplicity and visibility. Visibility is the idea that commands/instructions in the programming language should result in changes within the machine which are understandable, which can be effectively described in terms of the basic components of the machine. That is, a computer should not be seen as a black box which, when fed with some input, gives back some output. Rather, it should be seen as a number of basic components, interacting with each other, that are modified in precise, definable ways when instructions are executed.

Simplicity of the model is also very important. Unnecessary details of the underlying machine should be omitted, depending on the users' needs. If students are not introduced to the concept of pointers, there is no need to introduce the idea of variables not having symbolic names, i.e. the concept of an address. All that may be required is to introduce variables as being like boxes, with names, in which values can be put and retrieved.

Conceptual model as an advance organizer

The use of a conceptual model can be seen as a form of advance organizer. An advance organizer is

> ... (introductory material at a high level of generality and inclusiveness presented in advance of the learning material) whose relevance to the learning task is made explicit, to serve an assimilative role, rather than to rely on the spontaneous availability or use of appropriate anchoring ideas in cognitive structures. [2-p.131,132]

Mayer [26,28] showed that a conceptual model could play the role of an advance organizer. Three groups of non-programmers were presented with material on programming. The first group was shown a conceptual model of the machine before being presented with the material. The second group had the model shown to them after they had read the material. The third group was not presented with the model at all. The first group was shown to perform better on problems that required transfer of knowledge, i.e. problems which were slightly different from the one presented in the examples, and on problems of interpretation, i.e. finding the output of a program given some input. On the other hand, students of the second and third group performed better in problems that required generating programs similar to the ones that were presented in the material.

An important fact was that there was no significant difference between the performance of the second and third group; learning about the model after the presentation of the material did not help students at all. Being presented with the model before reading the material allowed the students to relate what they read to this model; the model served as background knowledge to

which the material could be related.

> When appropriate models are used, the learner seems to be able to assimilate each new statement to his or her image of the computer system. [28-p.133]

Another interesting fact is that "poorer" students benefited more from the use of a model than "better" students. This might simply mean the latter already had such a model or had managed to construct one from the material.

Conceptual models and analogies

The use of analogies is very important in learning a new subject, as it creates a basis on which to anchor new ideas. This anchoring is obtained by relating the new ideas to the old ones, using some kind of homomorphism.

There are many analogies that can be useful in understanding the basic principles underlying the functioning of a computer. However, there are dangers in using analogies. Some aspects of the new situation might not be explainable in terms of the old one. The use of the analogy is then limited. Furthermore, some aspects of the old situation might be assumed to hold in the new one, when they do not.

This last aspect is the more dangerous one. If students are told that something is analogous to another domain which they know, they might then try to reason in the new domain by using the properties of the old domain. As long as those properties are satisfied, there are no problems. But when properties do not hold in the new domain, errors may arise.

As Halasz and Moran explain in "Analogy considerer harmful" [15], analogies should be used exclusively as literary metaphors, that is, as a way of giving an intuitive idea of the concept being introduced. Analogies should not be used for reasoning with the ideas/concepts of the new domain.

> Literary metaphor is simply a communication device meant to make a point in passing. Once the point is made, the metaphor can be discarded. [15-p.385]

To reason with the new domain, a conceptual model should be used instead.

Conceptual model and virtual machine

This idea of associating a model of a machine with the constructs of a programming language is a very important one:

> A programming language is more than a notation for giving instructions to a computer. A language and the software that "understands" it can totally remake the computer, transforming it into a machine with an entirely different character. ... A new language brings with it a new model of the machine. [45-p.70]

This idea of a virtual machine can also be related to the process of structured programming. Dijkstra views the decomposition of a structured program as the definition/description of a series of virtual machines which are successively refined, implementing a higher-level virtual machine with the operations of a lower-level one, and repeating this process until the operations can be directly implemented using the operations associated with the programming language being used.

> I want to view the main program as executed by its own, dedicated machine, equipped with the adequate instruction repertoire operating on the adequate variables and sequenced under control of its own instruction counter, in order that my main program would solve my problem if I had such a machine. [5-p.48]

This is a very important process, which represents a powerful abstraction mechanism:

> For me, the conception of this virtual machine is an embodiment of my powers of abstraction, not unlike the way in which I can understand a program written in a so-called higher level language, without knowing how all kinds of operations (such as multiplication and subscription) are implemented and without knowing such irrelevant details as the number system used in the hardware that is eventually responsible for the program execution. [5-p.48,49]

4.2.3. The spiral approach

> Informal analyses of universities which have adopted the spiral approach suggests that the drop-out rate decreases and that students are actually willing to do more work because of the positive reinforcement of successful mastery. The spiral approach seems to have been most helpful to the poorer students who were overwhelmed by previous pedagogic strategies. [35-p.196]

The spiral approach can be seen as the antithesis of the approach used in some programming language book where the learning material is organized into "discrete compartimentalized sections, each one containing all information relevant to a particular aspect of the language" [22-p.310]. Instead, the underlying basis for the spiral approach is

> ... to present students with a small amount of syntactic and semantic knowledge which can be "anchored" to their "ideational structure".
>
> ... At each step the new material should contain syntactic and semantic elements, should be a minimal addition to previous knowledge, should be related to previous knowledge, should be immediately shown in relevant, meaningful examples and should be utilized in the student's next assignment. [35-p.195]

Of course, this means students will be introduced to the same topic more than once, each time in a more complex and detailed way. In many ways then, the spiral approach is to teaching as top-down stepwise refinement is to programming.

5. Teaching programming at the University of Waterloo

In this chapter, we examine what is done at the University of Waterloo, in the introductory programming courses. More precisely, we examine the textbooks [7,8] which are used in the courses CS118 and CS140. [24] First, a general evaluation of these textbooks is given, using the elements introduced in the previous chapter. Then, some of the basic programming scripts employed in the books are identified.

---

[24] Two other introductory courses are also offered: CS115, which is for business students and CS116, which is for social science students. As their respective textbooks spend little time on the subject of programming, we

Note that the main difference between these two courses is in the intended audience. CS118 is intended for science students, whereas CS140 is for mathematics students. They cover the same material, except that the programming assignments in CS140 are more involved [9].

5.1. General overview

The first remark about the two textbooks is that, except for the programming language they use, which is FORTRAN/77 in one case [8], and PASCAL in the other [7], their content and structure are the same. In the following, all references will be to [8] as it is currently the textbook used in CS118, and in one of the sections of CS140 [9].

An important characteristic of the book is the distinction which is made between an algorithm and a program. In chapter 2, a pseudocode language is introduced to express algorithms. It consists of the following instructions: **get, put,** assignment operation, **if-then-else, while-do** and **stop.** Additional facilities are subsequently added: **for** loop, procedures and functions (called modules), arrays. Many examples of algorithms are then given.

In the next chapter, the FORTRAN/77 programming language is introduced, and rules for translating the pseudocode into FORTRAN/77 are given. All programs shown in the text are always obtained from the translation of pseudocode developed for solving some particular problem.

During the first few weeks of class, students are not expected to effectively program on the computer. Their first assignments are concerned only with the basic concepts of algorithms. This seems like a good solution for avoiding *computer shock*. When students finally access the computer, they should already have mastered the concepts of elementary algorithms.

The textbook also exemplifies the spiral approach in the way the notion of algorithm is presented. In chapter 1, the intuitive idea of algorithm as a problem-solving procedure is first presented. Then, chapter 2 introduces more explicitly this notion by defining a pseudocode in which algorithms can be expressed, independently of any programming language. Chapter 4 then presents some elementary algorithmic techniques, and new algorithms are developed. An interest-

---

do not examine them.

ing point is that the examples are developed in a structured manner, using top-down stepwise refinement. The next chapters (6, 7 and 8) then use what has been presented so far in solving many new problems. In chapter 9, students are then introduced formally to the topic of algorithm design using a structured, top-down stepwise refinement process. By this time, they have a better understanding of what algorithms can do, and can now benefit from a more formal and detailed approach.

The presentation of sorting algorithms also reflects such a spiral approach. Sorting by exchange is presented for the first time in chapter 5, for sorting two, and then three numbers; the latter problem incorporates the solution to the previous problem. Sorting techniques are then examined in more details in chapter 13, once arrays have been discussed

As its subtitle indicates, this book really is an introduction to structured problem solving, more precisely to mathematical problem solving. A good number of exemplar problems are given and solved in great detail. Also, new problems are often solved using algorithms (modules) introduced in previous chapters, showing the importance of a correct problem decomposition.

One criticism, however, is that, although only basic elements of pseudocode are introduced in the second chapter, they are presented without reference to any computer model. For example, variables are introduced in the following manner:

> A fundamental idea in the algorithm is the use of the symbols *c* and *f*. In pseudocode such quantities are called *variables*. Unlike algebra, variables are used to represent known numeric values.
>
> ... Variables are *undefined* until they have been assigned a value. An undefined variable may not be used in an arithmetic operation. Furthermore, a variable can only have one numeric value at any given time. Once a variable has been given a value, it retains that value until it is changed explicitly. [7-p.9]

This way of defining a variable might not sound obvious to a novice programmer having no idea of what a computer is. Perhaps the general structure of a computer could be given, which would allow for a simpler definition of a variable. Or the following analogy could be used: A vari-

able **X**, can be seen as a box, identified by the name **X**. When the algorithm starts, all boxes/variables used are empty. Assigning a value to a variable **X** is like putting that value in the box named **X**. Using a variable in an arithmetic operation is like retrieving the content of the box. If the variable has never been assigned, i.e., the box is empty, it does not make sense to use it in an operation.

Some form of pragmatic knowledge is also presented. For example, on page 14, the set of operations required when using a **while-do** loop are explicitly mentioned:

> In order to use **while-do** actions in an algorithm, however, more information must be specified. In fact, each **while-do** has associated with it three important operations. Each operation is *essential* for the **while-do** action to function correctly. The operations are: 1)The initialization of the **while-do** control variables in actions preceding the **while-do** itself. (The *condition* cannot be evaluated unless its control variables have values). 2)The *condition* evaluation which controls termination of the **while-do** repetition. 3) The action sequence S which must change at least one **while-do** control variable. (If no control variable is ever changed by the action sequence, the **while-do** will never terminate.) [7-p.14]

Another example of pragmatic knowledge is in section 4.2.4: *End-of-Data Techniques - Counters and Flags*. The book describes two techniques for reading data. The first case is when the number of items to read is known in advance [7-p.73]:

```
* Acquire the number of items to process
get number
* Process each of a number of items
count ← 1
while count ≤ number do
    * Acquire and process a data item
    count ← count + 1
```

The second case is by using a sentinel value [7-p.74]:

* Acquire the first data item

get item

* Process items until the sentinel is encountered

while item ≠ sentinel **do**

    * Process the current data item

    * Acquire the next data item

    get item

On the other hand, the presentation of the **if-then** action could benefit from a more pragmatic approach.

> There is a second form of the **if-then-else** which is used when the action sequence S2 is null. This means that the **else** portions may be omitted, resulting in an **if-then** action. The general form of the **if-then** statement is given as follows,
>
> **if** condition **then**
>
> S1
>
> where condition and S1 are the same as for the **if-then-else** action. In the operation of an **if-then** statement, the *condition* is evaluated. If the result is **true,** action sequence S1 is executed, then statements following the **if-then.** However, if the result is **false,** execution continues immediately with the statement following the **if-then.** [7-p.23,24]

This is a syntactic definition of the **if-then** statement. It does not give the student any idea of when and why to use it, as opposed to an **if-then-else.** (What does it mean, practically, for an action sequence to be null?) Wouldn't it be better to give examples of the type of scripts which use this construct? More about this in the next section.

The only other critique might be that it is too mathematically oriented, which makes it difficult to use for people not having the appropriate background.

5.2. Programming scripts and plans

The following describes some of the scripts and plans which were extracted from the many algorithmic examples found in *FORTRAN/77 - An Introduction to Structured Problem-Solving* [8]. Note that this list is far from complete and does not include strategic plans.

Variable plans

Variable plans are used to describe certain roles played by variables; that is, they describe some of the pragmatic differences that exist between variables.

[1] **Frame**: Variable Plan

* Description: ...
* Where used?: ...
* Initialization: ...
* How used?: ...
* Type: ...

[1.1] **Frame**: New_Value_Variable Plan

* Description: holds value produced by generator
* Where used?: ...
* Initialization: ...
* How used?: as argument to producer function
* Type: ...

[1.1.1]

Frame: Read_Variable Plan

* Description: receives and hold a newly READ variable

* Where used?: [p.74,76,102,210,294]

* Initialization: **get** new_value

* How used?: **get** new_value

* Type: ...

[1.1.2]

Frame: Counter_Variable Plan

* Description: counts occurrence of an action

* Where used?: [p.73]

* Initialization: Counter := 0

* How used?: Counter := Counter + 1

* Type: Integer

[1.1.3]

Frame: Iterator_Variable Plan

* Description: Used to produce a sequence of values to be processed

* Where used?: [p.84,101,111,127,156,213,235,262]

* Initialization: Iterator := Initial Value

* How used?: Iterator := Succ(Iterator)

* Type: ...

[1.2] **Frame**: Temporary Variable Plan

* Description: Used to hold result temporarily

* Where used?: ...

* Initialization: unnecessary

* How used?: ...

* Type: ...

[1.2.1]

Frame: Temporary_Exchange Variable Plan

* Description: Used to hold value to perform an exchange

* Where used?: [p.113,318,332,414]

* Initialization: unnecessary

* How used?: Temp := First_value ... Second_Value := Temp

* Type: ...

[1.2.2]

Frame: Optimizing_Temporary Variable Plan

* Description: Used to avoid repetitive evaluation of a unmodified expression

* Where used?: [p.101,135,256,264]

* Initialization: unnecessary

* How used?: Temp := Expression; ... then used anywhere expression is needed (e.g. **while** (condition using Expression) **do**)

* Type: ...

[1.2.3]

Frame: Simplifying_Temporary Variable Plan

* Description: Used to hold the value of a complex expression to simplify the presentation.

* Where used?: [p.127,132,228,235,255,284]

* Initialization: unnecessary

* How used?: Temp := Expression; then used anywhere expression is needed (e.g. Temp := Expression; ... ;**put** Temp)

* Type: ...

[1.3] **Frame**: Indicator Variable Plan

* Description: Used to keep track of special value. Usually modified only by assigning a new value, not by an arithmetic operation.

* Where used?: ...

* Initialization: ...

* How used?: Indicator := Special_Value

* Type: ...

[1.3.1]

Frame: Flag Variable Plan

* Description: Indicates some state of the computation

* Where used?: [p.84,101,128,305,318,352]

* Initialization: Flag := true/false (Default)

* How used?: Flag := false/true (Reset)

* Type: boolean

[1.3.2]

Frame: Best_Value Variable Plan

* Description: Keep track of current special value which may later be updated when a new value satisfying some condition better is found, e.g. for searching for minimum/maximum.

* Where used?: [p.283,290,314,358,414]

* Initialization: Best_Value := Worst Value so far (which can be any value)

* How used?: Best_Value := New_Value

* Type: ...

[1.3.3]

Frame: Position_Indicator Variable Plan

* Description: Usually used in conjunction with a Best_Value Variable to keep track of its position when the latter is an array component.

* Where used?: [p.283,314,414]

* Initialization: Position_Indicator := Position(Initial Best_Value)

* How used?: Position_Indicator := Position(Best_Value)

* Type: Array_pointer

[1.4] **Frame**: Array_Pointer

* Description: Used to scan the elements of an array

* Where used?: [p.273,276,284,290,305,314,318,322,352]

* Initialization: Array_pointer := position within the range of the array

* How used?: Array_pointer := new position within the range of the array

* Type: ...

[1.5] **Frame**: Running_Total Variable Plan

* Description: builds up a value one step at a time

* Where used?: [p.235,273,276,284,394,403]

* Initialization: Running_Total := 0

* How used?: Running_Total := Combine(Running_Total, New_Value)

* Type: ...

Non-looping plans

The following are programming scripts involving no repetitive actions. Nonetheless, they may be used as components within looping plans. In other words, they are some of the basic scripts from which more complex ones can be built.

[2] **Frame**: Non_loop plan

* Description: Plan involving no repetition. Simply a sequence of actions.

* Where used?: ...

* Variables: ...

* What is tested?: ...

* Actions: ...

[2.1] **Frame**: Exchange Plan

* Description: Exchange the value of two variables

* Where used?: [p.113,317,332,414]

* Variables: Temporary_Exchange Variable, two other variables

* What is tested?: nothing

* Actions:

```
Temp := First_Var;
First_Var := Second_Var;
Second_Var := Temp
```

[2.2]

Frame: Avoid_Repetitive_Evaluation Plan

* Description: avoid the unnecessary reevaluation of a complex expression

* Where used?: [p.101,135,256,264]

* Variables: Optimizing_Temporary Variable

* What is tested?: nothing

* Actions:

Temp := Expression;

...

while (condition using Temp) **do**

...

[2.3] **Frame**: Simplify_Presentation Plan

* Description: Simplify the presentation of an expression by representing some subexpression by a single variable

* Where used?: [p.127,132,228,235,255,284]

* Variables: Simplifying _Temporary Variable

* What is tested?: nothing

* Actions:

Temp := Expression_1;

...

Var := Expression involving Temp

[2.5] **Frame**: Reset_Invalid_Value Plan

* Description: An invalid value/argument is replaced by a valid one

* Where used?: [p.241,484]

* Variables: New_value

* What is tested?: New_Value

* Actions:

if New_Value is invalid **then**

New_Value := Valid value

[2.5] **Frame**: Indicate_Invalid_Value Plan

* Description: Indicate an invalid value, otherwise, process it

* Where used?: [p.135,156,161,228,255,353]

* Variables: New_Value ot Flag

* What is tested?: New_Value (or Flag)

* Actions:

if value is invalid **then**

print error message (or set Flag_Indicator)

else

process the value

Looping Plans

The following programming scripts are concerned with iterative plans for solving problems.

[3] **Frame**: Loop Plan

* Description: ...

* Where used?: ...

* Variables: ...

* What is tested?: ...

* Set Up: ...

* Actions in body: ...

* Conclusion: ...

[3.1] **Frame**: Array_Operation Loop Plan

* Description: Manipulated all the elements of an array in a unique fashion. Operation can be: **get, put**, initialization, etc...

* Where used?: [p.273,286,352,475]

* Variables: Array, Array_Pointer

* What is tested?: Array_Pointer for out of range

* Set Up: Array_Pointer := Lower_Bound(Range(Array))

* Actions in body:

```
Operation( Array[ Array_Pointer ] );
Array_pointer := Succ( Array_pointer )
```

* Conclusion: ...

[3.2] **Frame**: Find_Satisfactory_Value Loop Plan

* Description: Search through an array looking for an element which satisfies a given condition

* Where used?: [p.305,309]

* Variables: Array, Array_pointer

* What is tested?: Array[Array_Pointer] for satisfaction of the condition

* Set Up: Array_Pointer := Lower_Bound(Range(Array))

* Actions in body: Array_pointer := Succ(Array_Pointer)

* Conclusion: determine which condition ended the loop, i.e. a value was found or the Array_Pointer became out of range.

[3.3] **Frame**: Find_Best_Value Loop Plan

* Description: search through an array to find the element satisfying in the best way a given condition, e.g., minimum/maximum

* Where used?: [p.283,290,314,358,414]

* Variables: Array, Array_Pointer, Best_Value, Position_Indicator

* What is tested?:

Array[Array_Pointer] vs Best_Value_Indicator

Array_Pointer for out of range

* Set Up:

Array_Pointer := Lower_Bound(Range(Array));

Best_Value_Indicator := Worst_Possible_Value (or Array[Array_pointer])

* Actions in body:

if Array[Array_pointer] better than Best_value

then

Best_Value := Array[Array_Pointer];

Position_Indicator := Array_pointer;

endif

Array_pointer := Succ(Array_pointer)

* Conclusion: ...

[3.4] **Frame**: Counter_Controlled Running_Total Loop Plan

* Description: Builds a running total in the loop, with iteration controlled by the value of Counter

* Where used?: [p.274,282,353,394,402]

* Variables: Counter, Running_Total, New_Value

* What is tested?: Counter

* Set Up: Initialize variables

* Actions in body: Test, Read, Count, Total

* Conclusion: ...

[3.5] **Frame**: Sequence_Generator Loop Plan

* Description: a sequence of values is generated and each is used in computation to produce a sequence of values for an expression, e.g. for tabulation of a function, or generation of all primes between certain bounds. Optionally, the sequence generation can be aborted.

* Where used?: [p.84,111,127,132,138,213,235,256,264]

* Variables: Iterator (Flag optional)

* What is tested?: Iterator (and Flag) for end of sequence

* Set Up: Iterator := Initial Value of sequence (Flag:=true)

* Actions in body:

    Process current value; (if necessary, reset flag to abort)

    Iterator := Succ(Iterator)

* Conclusion: determine if the computation was completed or aborted

Other scripts exist and would have been appropriate for inclusion in this section. However, the above are sufficient to show that some form of pragmatic knowledge can be made explicit. If this knowledge was transmitted explicitly, students would surely benefit from it.

6. Conclusions

The type of knowledge necessary for programming is very complex. Although knowledge of programming languages is an important component of programming knowledge, it is not sufficient. Knowledge of problem-solving strategies and some form of pragmatic knowledge is also required.

The main task in learning any complex domain of knowledge is the acquisition of the basic paradigms of the field. Learning the basic paradigms of computer science requires the acquisition of many different types of knowledge. One of the main tasks of teaching computer science should therefore be to assist students in acquiring this knowledge and hence, the necessary paradigms.

By introducing students to the conceptual model of a machine, by using a spiral approach, and by introducing programming scripts explicitly, we can facilitate the acquisition of this knowledge. The University of Waterloo, in many of these aspects, adheres to these educational principles, although students might benefit from a more explicit presentation of the basic programming scripts.

Furthermore, making explicit the tacit knowledge that students have to acquire should not be restricted to introductory computer science courses. It should also be applied to advanced computer science education, as well as other domains of knowledge. Of course, this requires that this tacit knowledge be explicitly recognized and acknowledged. In the case of computer science, which is evolving so rapidly, this is even more crucial. An awareness of the paradigms one is employing, is a key factor in order to facilitate further growth.

7. References

[1] Adelson, B., "Problem-solving and the development of abstract categories", *Memory and cognition,* vol. 9, 1981, pp. 422-433

[2] Ausubel, D.P., *Educational psychology: A cognitive view,* Holt, Rinehart and Winston, 1968

[3] Capra, F., *The TAO of physics,* Flamingo, 1983

[4] Dijkstra, E.W., "The humble programmer", *CACM,* vol. 15, no. 10, 1972, pp. 859-866

[5] Dijkstra, E.W., "Notes on structured programming", in Dahl, O.-J., Dijkstra, E.W. and Hoare, C.A.R., *Structured programming,* Academic Press, 1972, pp. 1-82

[6] DuBoulay, B., O'Shea, T. and Monk, J., "The black box inside the glass box: presenting computing concepts to novices", *IJMMS,* vol. 14, no. 3, 1981, pp. 237-249

[7] Dyck, V.A., Lawson, J.D., Smith, J.A. and Beach, R.J., *Computing - An introduction to structured problem solving using PASCAL,* Reston Publishing co., 1982

[8] Dyck, V.A., Lawson, J.D. and Smith, J.A., *FORTRAN/77 - An introduction to structured problem-solving,* Reston Publishing co., 1984

[9] Dyck, V.A., *Private communication,* December 1984

[10] Floyd, R., "The paradigms of programming", *CACM,* vol. 22, no. 8, 1979, pp. 455-460

[11] Gabrini, P., "Integration of design and programming methodology into beginning computer science", *SIGCSE Bulletin,* vol. 14, no. 1, 1982, pp. 85-87

[12] Greeno, J.G., "The structure of memory and the process of problem-solving", in Solso, R. (ed.), *Contemporary issues in cognitive psychology,* Winston, 1973, pp. 103-133

[13] Greeno, J.G., "Cognitive objectives of instruction: theory of knowledge for solving problems and answering questions", in Klahr, D. (ed.), *Cognition and instruction,* Lawrence Erlbaum Associates publishers, 1976, pp. 123-159

[14] Gruener, W.B. and Graziano, S.M., "A study of the first course in computers", *SIGCSE Bulletin,* vol. 10, no. 3, 1978, pp. 100-105

[15] Halasz, F. and Moran, T.P., "Analogy considered harmful", *Proceedings of the conference on human factors in computer systems,* Gaithersburg, Maryland, 1982, pp. 383-386

[16] Hanson, A. and Maly, K., "A first course in computer science: what it should be and why", *SIGCSE Bulletin,* vol. 7, no. 1, 1975, pp. 95-101

[17] Hayes-Roth, F., Waterman, D.A. and Lenat, D.B. (eds), *Building expert systems,* Addison-Wesley, 1983

[18] Houston, J.P., *Fundamentals of learning and memory,* Academic press, 1981

[19] Howe, J.A.M., "Developmental stages in learning to program", in Klix and Hoffman (eds), *Cognition and memory,* 1980, pp. 253-263

[20] Hyde, R.C., Gay, B.D. and Utter, D., "The integration of a problem-solving process in the first course", *SIGCSE Bulletin,* vol. 11, no. 1, 1979, pp. 54-59

[21] Kernighan, B.W. and Plaugher, P.J., *The elements of programming style,* McGraw Hill, 1974

[22] Kreitzberg C.B and Swanson, L., "A cognitive model for structuring an introductory programming curriculum", *Proceedings of the national computing conference,* AFIPS press, 1974, pp. 307-311

[23] Kuhn, T.S., *The structure of scientific revolutions,* vol. II, no. 2, The University of Chicago Press, 1970

[24] Langley, P. and Simon, A., "The central role of learning in cognition", in Anderson, J.R. (ed), *Cognitive skills and their acquisition,* 1981, pp. 361-380

[25] Mayer, R.E., "Information-processing variables in learning to solve problems", *Rev. Educ. Res.,* 45, 1975, pp. 525-541

[26] Mayer, R.E., "Different problem-solving competencies established in learning computer programming with and without meaningful models", *J. Educ. Psychol.,* 67, 1975, pp. 725-734

[27] Mayer, R.E., "A psychology of learning BASIC", *CACM,* vol. 22, 1979, pp. 589-593

[28] Mayer, R.E., "The psychology of how novices learn computer programming", *Computing Surveys,* vol. 13, no. 1, 1981, pp. 121-141

[29] Olderhoeft, R.R. and Roman, R.V., "Methodology for teaching introductory computer science", *SIGCSE Bulletin,* vol. 9, no. 1, 1977, pp. 123-128

[30] Polak, W.H., *Compiler specification and verification,* Spinger-Verlag, LNCS-124, 1981

[31] Resnick, L.B. and Ford, S., *The psychology of mathematics for instruction,* Lawrence Erlbaum Associates publishers, Hillsdale, 1981

[32] Rumelhart, D.E. and Norman, D.A., "Accretion, tuning and restructuring: three modes of learning", in Cotton, J.W. and Klatsky, R.L. (eds), *Semantic factors in cognition,* Lawrence Erlbaum Associates publishers, 1978, pp. 37-53

[33] Rumelhart, D.E. and Norman, D.A., "Analogical processes in learning", in Anderson, J.R. (ed), *Cognitive skills and their acquisition,* Lawrence Erlbaum Associates publishers, 1980, pp. 335-359

[34] Schneider, G.M., "The introductory programming course in computer science - Ten principles", *SIGCE Bulletin,* vol. 10, no. 1, 1978, pp. 107-114

[35] Shneiderman, B., "Teaching programming: A spiral approach to syntax and semantics", *Computers and Education,* 1, 1977, pp. 193-197

[36] Shneiderman, B., *Software Psychology: Human factors in computer and information systems,* Winthrop Publishers, 1980

[37] Shneiderman, B. and Mayer, R., "Syntactic/semantic interactions in programming behavior: A model and experimental results", *Int. J. Comput. Inf. Sci.,* vol. 8, no. 3, 1979, pp. 219-239

[38] Soloway, E. and Woolf, B., "Problems, plans and programs", *SIGCSE Bulletin,* vol. 12, no. 1, 1980, pp. 16-24

[39] Soloway, E., Bonar, J., Woolf, B., Barth, P., Rubin, E. and Ehrlich, K., "Cognition and programming: Why your students write those crazy programs", *Proceedings of the national educational computing conference,* North-Texas State university, 1981, pp. 206-219

[40] Soloway, E., Ehrlich, K., Bonar, J. and Greenspan, J., "What do novices know about programming?", in Shneiderman, B. and Badre, A. (eds), *Directions in human-computer interactions,* ABLEX Inc., 1982, pp. 27-54

[41] Soloway, E., Ehrlich, K. and Bonar, J., "Tapping into tacit programming knowledge", *Proceedings of the conference on human factors in computing systems,* Gaithersburg, 1982, pp. 52-57

[42] Soloway, E., Bonar, J. and Ehrlich, K., "Cognitive strategies and looping constructs: An empirical study", *CACM,* vol. 26, no. 11, 1983, pp.853-860

[43] Soloway, E., "A cognitively-based methodology for designing languages / environments / methodologies" *Proceedings of the ACM SIGSOFT/SIGPLAN software engineering symposium on practical software development environments,* 1984, pp. 193-196

[44] Soloway, E., Talk given at the university of Waterloo, November 12th 1984

[45] Tesler, L.G., "Programming languages", *Scientific American,* vol. 251, no. 3, 1984, pp. 70-78

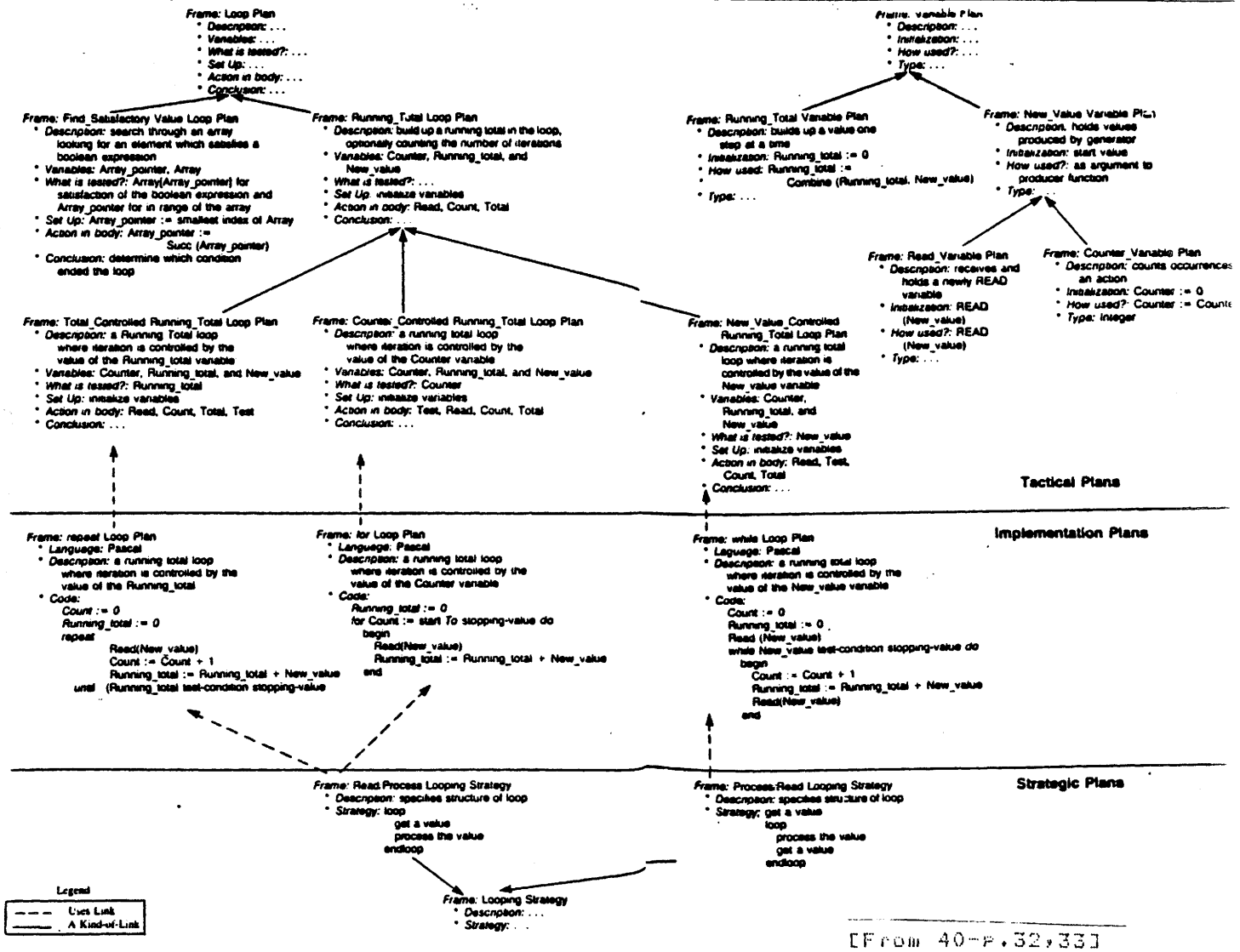
[46] Weinberg, G., *The psychology of computer programming,* Van Nostrand Reinhold, 1971

[47] Wessels, M.G., *Cognitive psychology,* Harper and Row publishers, 1982

[48] Wittrock, M.C., "Learning as a generative process", *Educational Psychologist,* vol. 11, no. 2, 1974, pp. 87-95

[49] Woodhouse, D., "Introductory courses in computing: aims and languages", *Computers and education,* vol. 7, no. 2, 1983, pp 79-89

Appendix 1



[From 40-P.32,33]