Learning a New Programming Language

Ronald Blaschke

Bowling Green State University

Computer Science

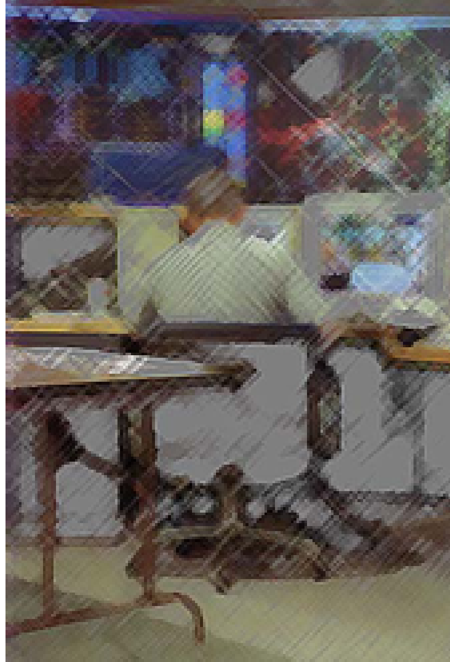
Bowling Green State University

Bowling Green, Ohio 43402

$Date: 2000/05/04 04:35:37 $

$Revision: 1.2 $ ($State: Final $); available online at

http://www.cosy.sbg.ac.at/~rblasch/study/cs665/essay/



Abstract

Learning a new programming language can be difficult and the pressure of "getting a job done" in an unknown language is even more difficult. But what are the common mistakes and how can they be avoided? And if you have the choice of giving the task to a novice and an experienced programmer, whom should you choose? Or does this not matter, since both of them don't know the new language? We will try to answer these and other questions and point out what is important when learning a new programming language.

Introduction

Almost every programmer is sooner or later confronted with learning a new programming language. Many computer science studies start with a procedural language, like Pascal or Ada, to show basic algorithms and data structures. Later on, for example in an operating systems course, one may have to use C or C++. The study may also include the introduction to a functional language, like LISP or Haskell, and to a logical language, like Prolog. Or a programmer may work on a project where the use of a specific language is required. Even if one wants to extend an editor like Emacs or Microsoft Word with macros a user may find himself learning a new language. In short, it is quite unlikely that a programmer

has to know only one language. We want to discuss some issues related to learning a new language and answer questions like: What are the most important cognitive factors for learning a new language? Is there a lot of negative transfer to the new language, i.e. are concepts learned in a former language transferred to the new language, but are there inappropriate? And what is of most help when learning a new language?

Structures and Plans

First, we need to introduce two terms that turn out to be important: Structure and Plans.

Structures

A structure describes the way something is build. For our analysis we distinguish between syntactic and semantic structure, as identified by Scholtz and Wiedenbeck ([2], [3]). The syntactic structure refers to the syntax of a programming language, for example the characters needed to start a comment.

The semantic structure refers to the semantic of a programming language, for example if a variable needs to be declared or not or if the function `read()` reads a whole line or a single character.

Plans

A plan is, in simple terms, how someone intends to do something. For our analysis we have strategic, tactical and implementation plans, as identified by Soloway et al. ([12]). The highest level of abstraction is the strategic plan. At this level we are language-independent and only concerned with the basic idea of doing something. An example would be the main loop of a server: Get request, process it and start over again. Even such high level things like incremental programming fall into strategic planning.

The tactical plan is more detailed, but still language-independent. It is a local strategy or algorithm for solving a problem.

Finally, the implementation plan is at the level of the programming language, i.e. the actual constructs a programmer would use to implement the language-independent tactical plans. Note that this is still somewhat above the syntax and the semantics we described above. Programmers don't worry about "should there be a semicolon or not," but for example think about "is there a loop construct to iterate over a list?"

For example, say we want to translate all uppercase characters to lowercase characters. A strategic plan may look like this.

```
loop
   read character from input
   translate character
   write character
```

Another example might be.

```
loop
   read line from input
   translate characters in line
   write line
```

A tactical plan for the first example is.

```
while (characters on input)
   read character
   if character is uppercase then translate to lowercase
   write character
```

And for the second example.

```
while (lines on input)
   read line
   translate all characters in line to lowercase
   write line
```

Further refinement leads to an implementation plan, e.g. for the language C.

```
char c;
while (c = getChar()) {
   print("%c", tolower(c));
}
```

Or for example the second in Perl.

```
while(<>) {print lc}
```

Of course, there are lots of plans to solve the problem.

Novice Programmers

As expected, novice programmers seem to have the hardest time learning a new programming language. Their programming knowledge is very much connected to the syntactic and semantic structure of the language (or languages) they know. Canas et al. ([4]) tested two groups of programmers, one with and the other without a debugging utility. Using a debugger was identified as enforcing a more semantic representation, while not using one a more syntactic. They concluded that both mental representations are equally effective.

The new structure of the new programming language seems to interfere with knowledge from the past. Scholtz and Wiedenbeck ([1]) tested a group of novice programmers and found that they often used the syntax of the former known languages (inexperienced programmers even used natural language). Extra training is needed to overcome this.

Syntax errors are quite hard for inexperienced programmers, who often find error messages to be unintelligible (Scholtz and Wiedenbeck [11]).

For more experienced programmers the syntax seems not to be a big problem. Many programmers who already know a programming language simply assume a certain pattern, mostly similar to the previously learned languages, as already mentioned. In other words, programmers have strong expectations about the

syntax. The reason why programmers go easy with the syntax seems to be the fact that a program will not compile with a syntax error. Also, the error messages are often quite clear to them. Problems with the syntax are easily resolved with a look in the reference manual (Scholtz and Wiedenbeck [11]).

Semantic errors, on the other hand, are different. A program may compile and even run with semantic errors. To make things worse, error messages which result at run time from semantic errors are not very easy to use to find the errors. As programmers become aware of this they tend to be more careful using them. They make sure that they understand the semantics of a construct first (Scholtz and Wiedenbeck [11]).

Nonetheless, the most important factors are the plans. As Rist ([9]) discovered, they seem to be available for recall after their first creation and tell us how our solution to a problem should look like. But each programming language has its own concepts, it's own style. If the unfamiliar concepts of the language are ignored suboptimal solutions occur and valuable features remain unused. This happens if one is trying to use the same plan as in a known language, or if the focus is on trying to get a working program rather than learning the language concepts [2].

Strategic plans are the most general, language-independent form of plans. Programmers worry very little about them, simply because they are so general. Some plans are so general that they could be applied to almost any problem, like

"It strongly suggests to write the program incrementally." (Scholtz and Wiedenbeck [11])

Tactical plans, which are basically the algorithms, are more frequent, yet still language-independent. The key point at this level is how many plans are available to a programmer, how many ways does he know to do something. Needless to say that this is a weak point of novice programmers.

Implementation plans deal with applying the language-independent tactical plans to the target programming language. It is easy when a construct is analogous to the construct of a known language. But things get complicated when a programmer can't figure out how to implement their tactical plan. If even the documentation can't help the programmers revise their tactical plans (Scholtz and Wiedenbeck [11]).

Even if a programmer is able to find a correct construct for what he is planning to do it is nonetheless possible that he uses the construct inappropriately (Scholtz and Wiedenbeck [11]).

This iterative interaction between tactical plans and implementation plans is important. It shows that programmers are not efficient in applying their past planning knowledge to solve problems (Scholtz and Wiedenbeck [11]).

Experienced Programmers

Under certain conditions experts may be as good as novices. Chase and Simon

([6]) performed an experiment where both, chessmasters and novices, had to remember valid and invalid configurations of chessboards. It turned out that the chessmasters were much better than the novices in remembering the valid chessboards, but they performed equally good for invalid boards. McKeithen ([7]) and Shneiderman ([8]) found that this applies to computer programs too. In short, the knowledge of experts may be nullified at some time [2]. The question is: Can this happen here too? Can a novice learn a new programming language as fast as an experienced programmer?

It doesn't seem so. Wu and Anderson ([5]) conducted an experiment where they had programmers who knew Pascal, Prolog or LISP. The experiment showed that someone who knows Prolog seems to learn LISP faster than someone who knows Pascal, and vice versa, simply because of the common elements of the languages, like recursion.

The knowledge of experienced programmers seems to be not as connected to a programming language as it is for a novice (Scholtz and Wiedenbeck [3]). This means that an experienced programmer can focus more on planning rather than the syntax and semantic of the language (Scholtz and Wiedenbeck [2]).

Experienced programmers seem also to be aware that the syntax of a language is essentially arbitrary. They know that it must be absolutely correct, or the program will not compile. Therefore they allocate time and attention to it. But again, they take it easy because the compiler will catch any errors.

The statements for novice programmers concerning semantics seem to hold for experienced programmers too. They tend to be careful using semantic structures because they know that these errors are hard to find. But it appears that, as in the case of syntax, programmers efficiently apply their past knowledge (Scholtz and Wiedenbeck [11]).

Strategic plans seem to have almost the same importance for experienced programmers than for novices. This may simply be because this level is that abstract.

Tactical and implementation plans are a major issue for novice programmers and are so for experienced. Moreover, the ability to generate alternative tactical plans has been identified as a characteristic of more experienced program designers (Scholtz and Wiedenbeck [11]).

Conclusions

So, what are the main points that are done wrong by programmers learning a new language? First, programmers use familiar, well-understood tactical plans when they begin to program in a new language. As a result, their tactical plans, and accordingly their programs, look more like a program in their known language, if at all. Sometimes the languages differ that much that the plans can't be implemented at all (Scholtz and Wiedenbeck [11]).

As Scholtz and Wiedenbeck observed, programmers don't concentrate on the

strengths and weaknesses of the new language, but worry about low-level issues like “are there arrays in the new language?”

When learning a new language, the first thing to do is to get the big picture how the language is supposed to be used. Then one should have a look at the most important aspects of the language. Seeking help from others who know the language or reading parts of the documentation seems to be a good idea. If you are writing the documentation point out the important features of the language at the beginning. The proper use of the language should be introduced as soon as possible.

Novices need some extra help in the beginning with the syntax and semantic of the new language. Experts, on the other hand, do not need this additional help, because their plans are often disconnected from a specific language. Although it should be fully documented for reference, it is a minor issue when learning a new language. The single exception is when there are constructs with different semantics in different languages. These constructs need to be pointed out.

Scholtz and Wiedenbeck ([11]) concluded in an experiment that 40% of their subjects’ efforts were spent for syntax and semantic problems, but these were resolved quickly and smoothly.

Plans and their creation are most important when learning a new language, for novices and experts. For example, was the language designed to work on lines or characters of input? Even if the language was designed for lines of input, working

on characters of input may also be possible because most languages are flexible enough. But the solution plans are suboptimal and non-idiomatic, i.e. the language is used improperly. Thus, focusing on the tactical and implementation plans seems to be a key to success.

Scholtz and Wiedenbeck ([11]) concluded, in the same experiment as before, that “the true source of difficulties lies in tactical planning and its interplay with implementation planning.”

So, when writing a tutorial or tutoring program (cf. Fix and Wiedenbeck [10]) the important aspects of the language should be highlighted. Showing the differences to another language, known by the programmer, makes it a lot easier for him to create the correct plans.

Fix and Wiedenbeck ([10]) did this by creating the tutoring tool ADAPT (Ada Packages Tool). The purpose was to help students that already knew Pascal or C to learn to use Ada packages. They do this by presenting a problem and several plans to solve it. If a programmer chooses a plan it is refined and several further ways to refine it are presented. The programmer thereby walks all the way down to the actual implementation of Ada code, thus learning to use Ada packages.

In order to further reduce the amount of negative transfer, i.e. transfer of concepts that are right in another language but are inappropriate in this one, they added buggy plans that are as said: Appropriate in another language (C or Pascal), but not in this one (Ada).

As Scholtz and Wiedenbeck ([11]) pointed out, a critiquing approach which started with presentation of how a problem might be solved in a familiar way and then shifted to a discussion of how it could be solved more effectively using the unique features of the new language might be most helpful.

If you want to take a single statement from this review with you I suggest the following.

> Scholtz and Wiedenbeck
>
> The main focus in teaching and learning second and subsequent programming languages should be on designing algorithms appropriate to the new language.

References

[1] Empirical Studies of Programmers: Fith Workshop, Edited by Curtis Cook, Edited by Jean Scholtz, Edited by and James Spohrer, 1993, 231 pages, 1-56750-088-9, Ablex Publishing Corporation, Norwood, An Analysis of Novice Programmers Learning a Second Language, Jean Scholtz and Susan Wiedenbeck, 1993, pages 187-205.

[2] Interacting With Computers, 1993, 130 pages, Butterworth-Heinemann Ltd., vol. 5, no. 1, “Using Unfamiliar Programming Languages”: The Effects on Expertise, Jean Scholtz and Susan Wiedenbeck, 1993, pages 13-30.

[3] International Journal of Man-Machine Studies, 1992, 130 pages, Academic Press, vol. 37, no. 2, “The Role of Planning in Learning a New Programming

Language", Jean Scholtz and Susan Wiedenbeck, 1992, pages 191-214.

[4] International Journal of Human-Computer Studies, 1994, 175 pages, Academic Press, vol. 40, no. 5, "Mental Models and Computer Programming", Jose Juan Canas, Maria Teresa Bajo, and Pilar Gonzalvo, 1994, pages 795-811.

[5] Workshop - Institut National de Rechereche en Informatique et en Automatique, 1992, Le Chesnay, no. 8, Knowledge Transfer among Programming Languages, Q. Wu and J. Anderson, 1992, pages 183-196.

[6] Visual Information Processing, 1973, Academic Press, "The Mind's Eye in Chess", William Chase and Herbert Simon, 1973.

[7] Cognitive Science, 1981, pages, Cognitive Science Society, Inc., vol. 5, "Knowledge Organisation and Skill Differences in Computer Programmers", Katherine McKeithen, 1981, pages 307-325.

[8] International Journal of Computer and Information Sciences, 1976, Plenum Publishing Corporation, vol. 5, no. 2, "Exploratory Experiments in Programmer Behavior", Ben Shneiderman, 1976, pages 123-143.

[9] Cognitive Science, 1989, 171 pages, Cognitive Science Society, Inc., vol. 13, no. 3, "Schema Creation in Programming", Robert Rist, 1989, pages 389-414.

[10] Computers Education, 1996, Elsevier Science Ltd., vol. 27, no. 2, "An Intelligent Tool to Aid Students in Learning Second and Subsequent Programming Languages", Vikki Fix and Susan Wiedenbeck, 1996, pages

71-83.

[11] International Journal of Human-Computer Interaction, 1990, Ablex Publishing, vol. 2, no. 1, "Learning Second and Subsequent Programming Languages": A Problem of Transfer, Jean Scholtz and Susan Wiedenbeck, 1990, pages 51-72.

[12] Directions in Human-Computer Interaction, Edited by A. Badre Edited by and B. Shneiderman, 1984, Ablex Publishing, "What do novices know about programming?", E. Soloway, K. Ehrlich, J. Bonar, and J. Greenspan, 1984.