Computer programming skills: A cognitive perspective

Graafsma, Irene

DOI:
10.33612/diss.168003240

Chapter 1

General Introduction

1.1 HISTORY OF PROGRAMMING AND DEVELOPMENT OF PROGRAMMING LANGUAGES

In the early 19th century, Joseph Marie Jacquard created a device to read punched cards that carried information on weaving patterns for the manufacturing of fabrics. His machine, combined with a traditional loom, simplified the process of manufacturing textiles by automating the creation of complex patterns (Essinger, 2004). This work formed the basis for the analytical engine created by Charles Babbage later that century, which consisted of columns that could be assigned different values for the mechanical machine to execute various mathematical computations (Dasgupta, 2014). The punched cards of Jacquard and the column structures of Babbage's machines were the first patterns designed for people to give instructions to a machine and could therefore be considered early forms of programming languages. Between 1842 and 1849, Ada Lovelace wrote the world's first published computer program, which consisted of a detailed method for calculating Bernoulli numbers with Charles Babbage's Analytical Engine (Fuegi & Francis, 2003).

Although these early machines and programs had a structured system designed for humans to communicate instructions to machines, these instructions were still limited to a single purpose, and were very abstract and technical. They did not resemble modern programming languages. In addition, despite these initial developments, it took almost a century for these types of instructions to be used more widely. Moreover, it was not until the 1940s that the first modern programming languages were created, that is, a full language-like system designed to communicate a wide range of instructions to a computer. However, these programming languages were still highly inefficient. Their use took a considerable amount of mental effort, because instructions had to be written in machine code directly (Knuth & Pardo, 1980). As machine code is the code that directly influences the computer hardware, a programmer was required to understand and instruct the exact steps that the computer hardware had to perform to execute a certain task. This required extensive hardware knowledge and very detailed and extensive code for operations that would today be considered very simple (Knuth & Pardo, 1980).

In the 1950s, there was rapid development of several programming languages, sparked by the development of the first compilers. A compiler is a program that translates instructions from a high-level language into the machine code that operates the computer

hardware directly, making it no longer necessary for programmers to write programs in complicated and inefficient machine code (Aho et al., 2007). In the 1960s and 70s, several higher-level programming languages were developed (e.g., Speakeasy, Smalltalk, C) that still form the foundation of most programming languages used today (Bergin & Gibson, 1996). In the 1980s, the focus was on developing programming languages to be more efficient and to allow for higher-level structures (e.g., classes for variables). In the 1990s, web programming gained importance and suitable "Rapid Application Development" languages, such as Java were developed (Bergin & Gibson, 1996). These later languages from the 90s and early 2000s were developed to increasingly resemble natural human languages (Fedorenko et al., 2019; Paulson, 2007).

With the creation of the first high-level programming languages also came the first programming education. The earliest programming languages were primarily developed to operate specific programs. Of course, they still had to be learned by the early programmers, but they were not suitable for general programming education. This made it difficult to learn to program, and only a select group of science and computing students engaged in such education programs. It was not until the 1980s, when the first computers became widely available, that there was a big spike in programming education, in primary and secondary school and for undergraduate students. This led to the first efforts to make programming languages more suitable for learners. To achieve this, three approaches were used: 1) mini-languages, these are languages specifically developed for learning, such as LOGO (Brusilovsky et al., 1997); 2) sub-languages, languages that select a subset of commands from a bigger language, such as Professor for Java (Gray & Flatt, 2003); and 3) an incremental approach, where a complex programming language is learnt by starting with just a few commands and then adding more and more complex commands (Hermans, 2020).

When click interfaces became more common in the 1990s and early 2000s, interest in widespread programming education seemed to diminish (HackerRank, 2018). However, over the last decade interest in programming education has started to rise again, with more employers demanding relatively advanced digital literacy from their employees in order to be able to customise analyses, applications or web pages (Rushkoff, 2012).

This recent increase in programming education has been accompanied by a renewed interest in programming as a skill. Since the 1990s, the field of computer science education has largely focused on different ways of teaching programming and the goals and motivations of learners (Guzdial, 2015). However, to teach programming optimally, there needs to be an understanding of which skills and traits underpin it. With a skill that is only 50 years old, which has evolved along with rapid development of different programming languages, requiring different skills, there is still much to discover in this area. This includes which skills or personality traits are important when learning to program in a modern-day context and how programming is processed in the brain. In order to start answering these fundamental questions about programming as a skill we need to study it from a cognitive perspective.

1.2 THE COGNITIVE PERSPECTIVE ON PROGRAMMING

The term “cognition” refers to the way the mind internally represents the external world and performs the mental computations required for all aspects of thinking. Cognitive science is a broad field of study focussing on the vast set of mental operations associated with such things as perception, attention, memory, language, and problem solving (Gazzaniga et al., 2009). The field uses a range of methods to study these processes, including studying the relationships between a target skill (like reading or programming) and specific underlying cognitive processes, the workings of the brain related to certain skills, and individual differences such as personality traits that may influence the acquisition of skills.

The cognitive perspective is particularly important with regards to programming education because it can tell us more about the nature of programming as a skill. This gives us the fundamental information necessary to teach programming, but also to predict who will be good at programming, and ultimately even to adjust programming languages themselves to be more suitable for the way people understand and learn.

Although the cognition of programming has not been the main focus of programming education research, there have been some important contributions in this area (Robins et al., 2019). Guzdial and du Boulay (2019) identified two main streams of cognitive research with different objectives. One stream focuses on whether cognitive

benefits accrue from learning to program (e.g., Pea & Kurland, 1984). The other stream researches which cognitive skills are important when learning to program. However, most studies in these streams were conducted in the 1980s and 1990s. Critically, since then the nature of both programming languages and of programming education has changed (Guzdial & du Boulay, 2019). At present, most computer science education research is performed by computer scientists and computer science education researchers, and less is conducted by psychologists and cognitive scientists. However, recently, the cognitive perspective has started to (re)gain more interest. Labs such as that of Professor Amy Ko at the University of Washington, have started to use an interdisciplinary approach to study topics such as programming language learning, and programming and problem solving. In particular they have contributed to the topic of programming and self-regulation (Xie et al, 2020) and to theories of programming instruction (Xie, Loksa et al., 2019). Other recent research (e.g., Prat et al., 2020) looks at cognitive skills and brain activity in relation to learning in a programming course. These researchers are starting to lay the foundations of modern-day cognitive research of programming. However, it is also clear that there are still many gaps regarding our understanding of the skills involved in learning to program (Prat et al., 2020; Xie, Loksa et al., 2019). In addition, not much is known about the neural processes involved in programming skills (Floyd et al., 2017; Prat et al., 2020; Siegmund et al., 2014). Even fewer studies have been performed focussing on autistic traits playing a role when learning to program (Wray, 2007). Therefore, it is interesting to study programming from a broad cognitive perspective in the modern-day context.

The studies in the current thesis fit into the second stream described by Guzdial and du Boulay (2019): they focus on which cognitive skills are important when learning to program, not on the benefits of learning it. They use two cognitive research approaches. The first is conducted in a programming education setting, aiming to learn more about the nature and subskills of programming by studying the skills and personality traits that predict learning success.

Within the first approach, the first step is to develop necessary methodology by adapting and validating an instrument to measure programming skill in university students (Chapter 2), specifically answering the research question:

(1) Can we create two reliable parallel short versions of the Second Computer Science 1 (SCS1) programming test?

This instrument is then used to test which cognitive skills predict programming success (Chapter 3), aiming to answer the research question:

(2) Which cognitive skills predict success at the end of a programming course?

The short versions of the SCS1 are also used to investigate which individual differences in personality, in the form of autistic traits, predict programming success (Chapter 4), focussing on the research question:

(3) Do autistic traits predict success in a programming course?

The second cognitive approach of this thesis is the use of neuroimaging to examine the brain activity related to programming and compare this to other skills for which the brain activity patterns are known. Specifically, in Chapter 5, Event Related Potentials (ERPs) are used to compare processing of a programming language and natural human languages in the brain, answering the research question:

(4) Is a programming language processed similarly to a natural language?

1.3 THEORETICAL FRAMEWORKS AND COGNITIVE MODELS OF PROGRAMMING

In more established cognitive research fields, such as memory and language, there are many theoretical frameworks that describe the processing that underpins these skills. Even for more specifically education-related skills such as reading and mathematics, several theoretical models are available on which to base research, allowing researchers to formulate explicit hypotheses and make specific predictions about potential outcomes. Examples of such models are the E-Z Reader and SWIFT models for reading (Rayner, 2009), and the problem-based learning model for mathematics (Mulyanto et al., 2018). In programming research, however, such models are still lacking.

In the 1980s and 90s, some models and frameworks were developed for problem solving in programming. These were mainly based on the results of studies that used think-aloud protocols where a participant is asked to verbalise their thought processes whilst solving a problem, such as Brooks' theory of the comprehension of computer programs (Brooks, 1977; 1978; 1983). A second range of models simulated the debugging process, such as the PUDSY system by Lukey (1980). More recently, Bednarik and Tukjainen (2006)

developed an eye-tracking model to characterise program comprehension processes. These models form an interesting start, however, none focus on the full programming process, but rather on a programming-related subskill, such as problem solving and debugging. Additionally, think-aloud studies typically use few participants and a limited number of tasks, because the reasoning process for each question has to be observed and analysed individually. Therefore, models based on these methods may be specific to that participant and task, and thus difficult to generalise. In cognitive science, researchers typically formulate both detailed models of subskills, and higher-level models that aim to provide broad generalisable understanding of a skill as a whole. These previous models of programming are lacking this broad overview and generalisability. Compared to reading and mathematics, programming is more complex, involving a wider range of subskills and, therefore, harder to model. However, in order to study programming skill in its entirety, a theoretical model of the full programming process is needed.

Recently, Armoni (2013) described a theoretical model of the programming process, called the PGK-hierarchy (named after Perrenet, Groote and Kaasenbrood who first defined it; Perrenet et al., 2005), which does aim to describe the programming process in general, regardless of the specific task or programming language (see Figure 1.1). According to the PGK-hierarchy, the programming process consists of four levels, each with a different purpose and different skills involved: The problem level, the object level, the program level and the execution level. At the problem level the programmer assesses the problem and the solution that is being asked for. Then, at the object level, an algorithm is formulated to solve the problem. This algorithm is not yet written out in a specific programming language, this happens at the next level, namely the program level. Finally, at the execution level, the computer interprets and executes the algorithm.

In the current thesis, the PGK-hierarchy is used to guide the predictions for the experiment described in Chapter 3, where we select cognitive skills that are possibly related to the first three levels of the model. I argue that the first two levels (the problem level and the object level) are related to algorithmic thinking or mathematical skills, while the third level, the program level, is related to natural language skills.

The relationship between language skills and programming performance has not been extensively tested, but several researchers have argued for a connection (Fedorenko et al., 2019; Hermans & Aldewereld, 2017). Throughout the current thesis, I explore this connection by looking at language skills as predictors of performance in a programming course; the relationship between learning success, language skills and autistic traits in programming students; and by comparing neural processing of syntax violations in a programming language, to grammar violations in programmers' first and second natural languages.

Figure 1.1. PGK-hierarchy model and related skills.

Level	Problem level	Object level	Program level	Execution level
Aim	Clarify problem and find solution	Translate high-level solution into an algorithm	Implement the algorithm in a specific programming language	The computer executes the code
Type of cognitive skills	Algorithmic thinking and mathematics		Language	No related cognitive skills

Note: This figure shows the four levels of the PGK-hierarchy model, first defined by Perrenet et al. (2005) and further described by Armoni (2013). The four levels are described in the top row, the aims of the levels as described by Armoni are listed in the second row. The third row indicates which type of cognitive skill I expect to play a role during each level of the hierarchy.

1.4 METHODS IN PROGRAMMING EDUCATION RESEARCH

Early programming education research in the 1980s and 90s mostly focussed on comparing novice to expert performance, specifically focussing on the number and different types of programming errors made by these groups. The focus was on overall programming performance, without relating this to underlying cognitive skills. The goal of these efforts was to optimise both teaching and the programming languages themselves (Guzdial & du Boulay, 2019). When the psychology of programming movement gained importance in the 1970s and 80s, the research focus shifted away from the programming languages and interface and put increasing focus on modelling the user. Over time, studies changed from think-aloud experiments with just a few participants in a laboratory to studies with larger

groups of learners and over longer periods of time in a more natural classroom setting (Guzdial & du Boulay, 2019). The teachers of these classes typically used a constructivist approach, where students were encouraged to learn programming languages by using them, without much direct instruction (Hermans, 2020; Portnoff, 2018). Recently, some researchers have argued that direct instruction, similar to the methods used in second language education and learning to read, are more beneficial (e.g., Hermans, 2020; Hermans & Aldewereld, 2017; Portnoff, 2018). However, this is still not the norm in today's programming education (Hermans, 2020). The students studied in the current thesis were still taught in a largely constructivist way. Another characteristic of the early classroom studies was that they used sample sizes that would today be considered small, usually consisting of a single classroom with 20 to 30 students per experiment. Today much larger samples are used in educational studies. Accordingly, the studies described in Chapters 2, 3 and 4 of this thesis are conducted with all the students from an undergraduate programming course, providing from 282 to 354 participants across the various chapters.

Another important development for the generalisability of the results of empirical research in modern computer science education was put forward by McCracken et al. (2001) who performed the first Multi-Institutional Multi-National (MIMN) computer programming research study. They emphasised the importance of testing programming skill and processes across different countries and institutions. This thesis acknowledges the importance of generalizability and implements the MIMN principle by further validating and using an independent programming test in Chapter 2, that has been used across countries and institutions, as well as by using this test in Chapters 3 and 4, thereby making the results easier to compare to existing and future studies that use the same instrument.

1.5 NEUROIMAGING METHODS IN PROGRAMMING LANGUAGE RESEARCH

In addition to examining which skills predict programming performance in education, the current thesis examines brain responses related to this skill. For natural language skills, such as reading and listening, it has been established how the brain responds to violations that are syntactic or semantic in nature.

Programming languages, however, have not yet been studied as extensively through neuroimaging. At the start of the current PhD project, a few studies examining

programming skills using functional Magnetic Resonance Imaging (fMRI) had been published (e.g., Floyd et al., 2017; Siegmund et al., 2014). The results of these studies suggest that areas that have been identified as being involved in natural language processing are activated during programming as well (Siegmund et al., 2014) and that brain activation patterns when reading a programming language are almost indistinguishable from the activation pattern during natural language reading once a programmer is sufficiently proficient in the programming language (Floyd et al., 2017). However, towards the end of the current PhD project, another fMRI study was published by Ivanova et al. (2020). They argue that reading and understanding programming languages is more reliant on the multiple demand system, which is the brain network most associated with mathematics, than on the language system. They also suggest that programming cannot be fully equated to either maths or language, but rather seems to rely on a novel cognitive network. However, their results also suggest that although they found that the language system was not activated in the same way for code as for prose, parts of the language system were still involved when reading code, particularly in the more language-based programming language Python. In addition, Prat et al. (2020) recently studied whether differences in resting brain activity were associated with success in learning programming. This study aimed to predict programming learning success in a university course from participants' performance on a number of cognitive tasks (including language tasks) and the students' resting state brain activity (measured with electroencephalography, EEG). They found that learning success in the programming course was mostly predicted by language skills, problem solving skills and working memory, while resting state EEG had little additional predictive value.

Although all four of these previous studies suggest that language skills play a role when programming, it remains unclear whether this means that the brain processes a programming language similarly to a natural language, and which elements of a programming language are similar or different to elements of natural languages. In order to directly study these differences we need to use a neuroimaging technique that allows us to measure responses to specific linguistic elements and compare those across languages.

One method to study brain responses to language is to use Event Related Potentials (ERP) in EEG signals. EEG, when measured in response to a specific stimulus, has a high

temporal resolution. While fMRI can tell us which brain areas are involved in a specific process, its temporal resolution is poor, making it difficult to link a response to, for example, a specific word in a sentence. EEG however, is very accurate in its timing, and can therefore be used to measure responses to specific words or symbols in spoken or written language. Typically, ERPs are used when studying natural languages and their structures (Carreiras & Clifton, 2004; Friederici et al., 2002). Thanks to research in many different natural languages we can make predictions about ERP-effects when a reader or listener is presented with a specific grammatical or semantic violation.

However, to date, we are not aware of any ERP studies examining violations in programming languages. Therefore, in Chapter 5, ERP responses to violations in a programming language (Java) are compared to those elicited by violations in the participants first and second natural languages. The results of this study directly contribute to our knowledge of programming languages by showing whether a violation in such a language elicits a similar response to a violation in a natural language. It also allows us to determine whether processing a programming language is more similar to processing of a second language than a first language.

1.6 CONTRIBUTION OF THE THESIS

With the four experimental studies described in this thesis, I aim to contribute to a better understanding of programming as a skill in modern society and education. The studies contribute to four different areas that are important in studying programming from a cognitive perspective: standardised measurements of programming skill (Chapter 2), cognitive skills that are predictive of programming performance in beginning programmers at the university level (Chapter 3), the relationship between programming and autistic traits (Chapter 4), and the use of brain responses to study similarities and differences between processing of programming language and natural human languages (Chapter 5). With the results of the studies in these four areas I aim to contribute to the existing knowledge on the cognition of programming and to lay foundations for future research. In the General Discussion (Chapter 6), I summarise and discuss the main methodological contributions and theoretical findings of the thesis, and present future directions for research in this field.