Using Programming Language Concepts to Teach General Thinking Skills

Martin Rinard

Department of Electrical Engineering and Computer Science
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 021139

1. Introduction

Programming languages have traditionally been seen as of interest only to software developers (who need to use languages to pursue their chosen profession). But the field has developed in response to an unprecedented situation in human history — the need to communicate with entities (i.e., computers) that are capable of carrying out very sophisticated computations involving prodigious amounts of information, but only if every aspect of the computation and information representation is specified explicitly and completely. Because of the need to make everything explicit and precise, concepts that arise in almost all human endeavors are present with unique precision, sophistication, clarity, and therefore accessibility. The availability of immediate, automated feedback through appropriate interaction with a computer enhances the accessibility of the concepts and promotes a fully rigorous approach that makes it difficult for a student to slide by with only a partial understanding. In this white paper I focus on three fundamental concepts:

- **Multiple Formal Perspectives:** Each programming language embodies a particular perspective on computation. Researchers have developed a wide range of languages, each with its own perspective. Because these languages are interactively explorable on a computer and have a precise semantics, studying programming languages is the ideal context in which to develop an understanding of 1) how different perspectives are appropriate for different problems and 2) that the perspective one employs can influence or even determine the outcome of a particular endeavor.
- **Automation and Translation:** Computers provide an unprecedented potential to automate tasks. Advances in automation correspond directly to advances in our ability to effectively manage the activities of our society. The drive to automate also fosters new, more general, and more productive ways of thinking about classes of problems.

  Much of the automation in programming languages deals with the translation of computer programs from one language into another. Advances in programming languages often correspond directly to advances in the automation of tasks previously performed by programmers. Automation and translation therefore provide an appropriate context for the exploration of which tasks should be automated and which should be left to humans. Understanding translation also provides students with a much deeper understanding of computation and an important systems building tool.
- **Abstraction:** Programming languages has produced a larger, more precise, and more sophisticated range of abstractions than any other field. It is therefore the ideal context in which to build an understanding of abstractions. This understanding is a core prerequisite for a successful career in computer science. It can also be one of the keys to understanding unstated assumptions in other fields, which in turn opens the door to new ways of conceptualizing and eliminating limitations in these fields.

In addition to providing reasoning skills and specific techniques that students will be able to draw on for their entire careers, a solid understanding of and ability to appropriately draw on these concepts can dramatically increase a person's ability to conceptualize, understand, and reason about various aspects of the intellectual and physical world in which we all live. Acquiring this understanding can therefore be an important (and in some cases even critical) component of the education of virtually any student, whether that student chooses to pursue a career involving any aspect of computer technology or not. Indeed, developing an understanding and mastery of these concepts in a rigorous context (such as the one that programming languages provides) may be more important for someone who pursues a career based largely on the humanities than for a practicing scientist or engineer. Through his or her daily work activities, a scientist or engineer will acquire at least a rough working understanding of the capabilities and limitations of formal reasoning as ap-

plied to his or her area of specialization. But someone who focuses exclusively on the humanities can easily fall into the trap of failing to appreciate or even recognize the need for a formal and precise approach when it is appropriate. An effective education can eliminate this shortcoming and enable the student to bring a deeper and ultimately more productive understanding to whatever endeavor he or she winds up pursuing.

2. Multiple Formal Perspectives

Each programming language provides a perspective — an identification of basic concepts that people use to formulate and solve problems in a domain of interest. Programming languages are typically designed to specify computations on a digital computer and therefore (potentially in combination with their implementation) have a completely precise formal semantics. Understanding that such a thing is possible and in some cases desirable is already an important intellectual leap. Students should acquire this understanding as part of an introductory class in computer science when they write computer programs.

The additional contribution that a programming languages class can make is understanding that there are multiple possible perspectives, and that the perspective one uses can make an enormous difference in the result one achieves. Specifically, the perspective affects and can even shape the following activities:

- **Problem Choice:** Because the perspective provides a way of thinking, it influences the problems one perceives and attempts to solve.
- **Problem Formulation:** Once one has identified a problem, the perspective plays an important role in how the problem is formulated — which parts of the problem one pays attention to and develops, which parts one ignores or even remains oblivious to.
- **Problem Solution:** There is a continuum of programming languages ranging from general-purpose languages (such as Java and C++) designed to support virtually any computation through to domain-specific languages designed to support only a very narrowly focused class of computations. There are also general-purpose languages (such as Prolog, Scheme, or ML) whose basic concepts and approaches differ significantly from other more popular general-purpose languages. It is well known that some languages work much better for solving certain classes of problems than others. When a language and a problem domain work well together, the solutions are more compact, easier to understand, and (as discussed further below) more reliable. Indeed, a large enough mismatch can make it impossible to satisfactorily solve a the problem.
- **Errors:** People working with a language and perspective that matches the basic concepts of the problem domain make many fewer errors than people working with a language that is less well matched. The reason is that working through a translation from the concepts in the problem domain to the concepts in the problem solution imposes a cognitive burden. Errors are often the result of this burden overwhelming the cognitive capacity of the person developing the solution. One of the reasons why programs written in general-purpose programming languages tend to have more errors than programs written in domain-specific languages is that general-purpose languages almost always impose a more complex concept translation than domain-specific languages (which ideally impose no concept translation at all).

2.1 Intellectual Importance

Understanding that perspectives can have such a powerful impact on results is an important realization. Most people adopt their perspectives unconsciously, to the point that much of the time they are not explicitly aware that they have a perspective at all. Knowing that alternate and potentially different or better perspectives are always available is a key stage in one's intellectual development. It helps people better understand the limitations of their reasoning, more readily listen to others that may have different perspectives, and integrate a multiple perspective approach explicitly into the way they choose, formulate, and solve problems. A person with this understanding can operate more effectively in a range of contexts that occur in all aspects of life, not just the professional activities of a computer scientist.

Given that this is such an important and general concept with such broad applicability, why is it best taught in the context of a programming language class (or, for that matter, in the context of computer science at all)? The reason is that the combination of programming languages and automated, immediate feedback via execution on a computer helps to make the concept precise, immediate, and accessible to students. For example, solving the same problem using two different languages with different (mis)matches to the concepts in the domain can make the importance of the perspective painfully obvious. This experience can prepare the student to then better comprehend the concepts associated with the presence of multiple perspectives when they are introduced as a topic of reflection after the exercise.

Activities in the humanities, in contrast, tend to be much more abstract, indirect, and imprecise. The feedback cycle is longer and the evaluation less clear cut. Because the evaluation is delayed and not automated, students can often cut corners and miss key details with no immediate consequences or feedback. The result is that students are less prepared to understand the significance of multiple perspectives, must make more of an intellectual leap to develop this understanding, and can more easily come away from the experience without having grasped the concept. Other fields of computer science and engineering tend to come with one specific perspective that works well in that field — indeed, one

of the primary purposes of education in that field is typically to transmit that perspective to students. Programming languages is close to unique in its heavy emphasis on multiple perspectives (as embodied precisely and concretely in different languages) as a central motif of the field. It is therefore the ideal field in which to explore this broad and general concept.

2.2 Pragmatic Applications

Understanding the importance of perspectives on solution effectiveness will be central to the success of students who choose to pursue careers in computer science or other areas with a strong technical component. Many tasks now involve the use of multiple programming languages, each with their own strengths and weaknesses, and understanding how to partition the problem across the languages is crucial to developing an effective solution. Even a problem as common as building a database-backed web service may involve the use of a scripting language, a database interface language, a traditional general-purpose programming language, and a hypertext preprocessor. Understanding how to operate effectively in these kinds of multilingual environments (and the potentially even more complex multilingual environments of the more sophisticated systems our students will develop in the future) will be crucial to the career success of our students.

One effective strategy is to organize the development of a system around a set of domain-specific languages engineered for that purpose. This approach can be dramatically more productive than the standard approach of using off-the-shelf languages that may not be as well suited to the task. This approach, however, requires a very talented developers. It is not clear how many of our students can be educated to this level of skill. There are two reasons why it is important to expose all students to this approach. First, it helps to ensure that students who do have the inherent potential to use this approach become exposed to the ideas they need to realize this potential. And second, it helps students who do not have this potential to be better able to work effectively in such an environment.

3. Automation and Translation

The emergence of automated systems with the capability, sophistication, and scale of modern computers is a landmark accomplishment in technology. The result is a dramatic increase in the inherent range of accomplishment potentially available to our species. But to realize this potential, we must understand which tasks to automate and which tasks are better left to humans. Note that this division will have a direct impact on our self-image as human beings and our conception of our place in the universe. Societies have a need to identify things that place humans apart. The discovery or development of other entities that can perform activities previously believed to be the exclusive province of humans can affect our perception of what it means to be human. Before the development of calculators, for example, the ability to do arithmetic quickly and accurately had great value and was perceived as something that set humans apart from other entities. A person could command respect in society because he or she had this ability. But as soon as a cheap automated alternative was available, this ability was devalued to the point that it was no longer considered to be a particularly valuable or distinctive human attribute at all.

3.1 Intellectual Importance

The development of every computer program has components that are today identified as human, valuable, and creative (identifying the problem to solve, creating the design to solve the problem, and building the solution) and others that are identified as appropriate for automation (the tasks associated with compiling the program into a executable representation). Moreover, the development of programming languages as a field has been driven in large part by the transfer of activities from the human domain to the automated domain. The emergence of high-level languages such as Fortran, for example, was predicated on the development of automated techniques such as allocating values to machine registers for computation and laying out data structures such as arrays in memory. Finally, there is an effective way to evaluate whether or not a specific division of tasks between humans and the automated system is productive — either the resulting system works or it doesn't.

3.1.1 Obtaining Automation

Programming languages therefore provides a rich context in which to explore questions relating to the division of tasks between automated systems and humans. The field supports historical approaches that look backward in time (how did certain tasks become automated and what was the impact of the automation), speculative approaches that look forward in time (what new programming language constructs can we develop with automated support in the implementation), and integrated approaches that combine the two. The precise nature of programming languages and computer programs allows questions to be posed with great precision. Because each language has an implementation, students can interact with the computer to quickly gain a comprehensive understanding of the automation and its value. Once students have explored questions related to the division of automation in programming languages, they are then prepared to see similar issues in other fields and work effectively to automate appropriate aspects of those fields.

3.1.2 Thinking Generally and Comprehensively

Automating various aspects of tasks also develops key thinking skills. In most fields practitioners and researchers deal with a single specific problem at a time. They therefore develop thinking skills oriented around obtaining insightful point solutions for clearly defined problem instances. But

to develop an automated system, one must think more generally and comprehensively to anticipate all of the important cases that can arise. This is a fundamentally different way of thinking. Mastering this kind of thinking can provide an enormous amplification effect as tasks that previously required human attention are shifted to an automated system. A common result is a dramatic increase in the scale of progress in the field — in some cases, it is possible to replace years of human effort with days or minutes of computing. Another advantage is the increased insight that comes from obtaining a more general perspective on the field. This general perspective can lead to new breakthroughs as the similarities between different problems become apparent and suggest new directions of understanding and exploration.

Programming languages is ideally placed as a field for exploring these issues. Existing languages provide the concepts and intellectual stimulation people need to automate various aspects. Moreover, the design of programming languages is the ultimate in general thinking — one is not simply using a building a system that automates some aspect of a given problem, but instead building a general tool that people can use to solve a variety of automation tasks. In this sense, the field of programming languages occupies a unique position within the entire field of human intellectual endeavor.

3.1.3 Planning Versus Improvisation

Translation inherently involves questions of when activities should happen. In almost every programming language implementation some activities take place before the program runs, which others take place only when the program runs. Understanding the tradeoffs involved in selecting where to place each activity can provide great insight into the greater issue of planning versus improvisation (two components of almost any human activity). Programming languages provides a productive educational environment for exploring this tradeoff, in part because it makes the trade-offs so obvious and easily accessible. A comparison of the time required to interpret a program versus the time required to run a compiled version provides an immediate and concrete illustration of the power of planning and setting things up ahead of time. On the other hand, understanding that other tasks simply cannot be performed without information that is available only when the program runs can also provide important insight. Finally, exploring the fluidity of moving tasks between compile time and run time (and the ramifications on programming language design) can provide insight into the advantages and disadvantages of planning and improvisation.

Embracing translation also provides a qualitatively deeper understanding of how to organize computation. Instead of simply having a program that executes, students can appreciate the more general concept of sequencing computation in stages as information becomes available over time. The whole concept of meta-computation — programs that manipulate other programs — introduces a new level of sophistication and helps students understand a much wider range of system structuring possibilities and techniques. Programming languages is the field with the greatest development of these ideas and therefore the field that is best placed to teach them.

3.2 Pragmatic Applications

Every computer science professional uses a compiler or preprocessor. Studying translation provides the concepts required to better understand and utilize the system in which the work takes place. Moreover, understanding how to write translators provides students with a powerful tool they can use to build systems more efficiently and effectively. For example, it enables them to build preprocessors, domain-specific languages, consistency checkers, and other tools that can improve their productivity and help them eliminate errors. It is important to understand that approaching development in this way, instead of simply using existing general-purpose approaches, can transform the development process, the cost of development, and both the functionality and quality of the resulting product.

4. Abstraction

Computer systems are tremendously complicated, arguably the most complex systems ever engineered. Abstraction (a principled discarding of detail) is therefore a central theme in virtually every field of computer science. But abstraction has developed most fully and completely in programming languages and is therefore most productively taught in the context of a course that focuses on programming languages. Consider the ways in which abstraction arises in programming languages:

- **Language Design and Features:** Most programming languages are abstractions of the underlying computing platform. The abstraction is made precise and, at some level, accessible via the implementation of the language.
- **Interfaces:** Each interface is an abstraction of the encapsulated functionality. Interfaces in general-purpose languages often provide a way to preserve some of the structure from the application domain in the final encoding of concepts from the domain in the program.
- **Specifications:** Specifications abstract the information that computations manipulate and the results and effects that the computation generates. Specifications are particularly interesting because they make the distinction between requirements and program behavior explicit and precise.
- **Types:** Types abstract the computations and information that the computation manipulates. Strictly speaking, types are merely a specific form of partial specification. However, types seem to hit a sweet spot in the specification space — they are often intuitive for programmers to provide, are efficiently and modularly checkable, and

have become the focus of a large part of the field of programming languages.

- **Program Analysis:** Program analyses work with abstractions of the state and computation. This abstraction is necessary to make the analysis tractable and able to operate without access to information that is available only when the program executes.

No other field comes close to providing abstractions with this level of variety, sophistication, and precision. In comparison with other fields, the precision of the abstractions is particularly well developed in programming languages. The concrete and abstract domains are often formally specified, as are the mappings between the two domains.

4.1 Intellectual Importance

Abstraction is a necessary process that most people engage in unconsciously. But an inappropriate abstraction can be problematic — if one discards important information during the process of abstraction, they can limit their thinking, acquire blind spots, or come to the wrong conclusion.

The explicit, overt, and precise nature of abstraction in programming languages can help students develop a general framework that they can use to improve their reasoning in other fields. This framework will help students perceive and understand both the need for abstraction and the potential problems that are an inevitable consequence of its use. Building on their experience with abstraction in programming languages, they will also be more able to see, understand, and criticize the implicit abstractions that they and others make in other fields where the abstractions are much less obvious and potentially even part of a standard and otherwise unquestioned approach. If students learn to apply a process of seeking out abstractions, understanding the resulting limitations, then, when appropriate, developing new abstractions that remove inappropriate limitations, they will be better able to understand how to innovate around shortcomings and challenge ways of thinking that have, over time, become limited or even counterproductive. When applied by someone with a sufficient command of the process, it can even become a mechanism for generating insights, creativity, and productive new directions upon demand.

4.2 Pragmatic Applications

Computer science professionals must deal with abstractions on a daily basis. Knowledge of programming language abstractions and how to use them is a prerequisite for functioning effectively in any professional environment. Moreover, many computer science professionals spend much of their time designing abstracts. A deeper understanding of abstractions in general can help them design more effective abstractions.

5. Potential Teaching Approaches

The field of programming languages is rich enough to support many productive teaching approaches. I outline two such approaches, but recognize that there are other approaches that may be equally or even more productive.

Both of the approaches start with software development activities. One advantage of this starting point is that it leverages the immediate feedback and precision that interacting with a computer provides and requires, which makes the concepts obvious and accessible. It is also possible to structure the software development activities to emphasize specific skills or approaches that students can immediately use in a professional software development career. It is also possible to take a more theoretical approach that starts with mathematical modelling or analysis activities. In either case, however, the most important part of the educational experience is reflecting on the concepts that the specific activities illustrate. This reflection is crucial to enabling students to generalize the concepts to apply them productively in different contexts, which is the most important and valuable lesson of the course. One way to incorporate this reflection into the course is to require students to perform additional assignments (typically in essay form) that demonstrate the application of the acquired concepts to a field outside of programming languages.

5.1 A Domain-Specific Language

This approach would organize the course around designing and implementing a language for computations in a particular domain. The course would be composed of several stages. The first would be to develop and understanding of the basic concepts in the domain. The second would be to design a language that supported the concepts explicitly. The third would be to implement the language. Finally, the students would implement several computations in the language.

Successfully developing the engineering artifacts would be only part of the educational process. The arguably more important part would be using the engineering experience as a foundation for reflecting on the different concepts. To this end students would write essays focusing on how their experiences relate to multiple perspectives, automation and translation, and abstraction in other contexts. Ideally, students would understand that the concepts in the domain-specific language were different from the concepts in general-purpose languages (the course would assume that previous introductory courses have already provided the students with knowledge of at least one such language) and that providing these concepts explicitly in the language provides a more productive perspective for computations in the domain than the perspective from general-purpose languages. Students would also understand that making the domain concepts explicit in the language shapes how they formulate and solve problems in the domain.

They would also understand how to generalize this example to understand that other perspectives would be useful in other domains, and indeed that some domains would be complex enough to justify the use of multiple perspectives. Finally, they would understand that it is possible (although potentially not ideal) to explicitly adopt a given perspective during the engineering of the system even though one may use a general-purpose language instead of a domain-specific language.

Students would address the automation and translation concepts as part of the implementation of the domain-specific language. They would understand that certain aspects of the implementation were automated by the implementation of the domain-specific language, and that these aspects would instead be performed by the human developer if the system were built in a general-purpose language. They would also understand how to generalize the basic concepts illustrated in this exercise to obtain an understanding of how automating tasks can make people more productive, that automation requires thinking about the general case as opposed to any specific single problem at hand, and that some tasks are still better left to humans. The implementation of the domain-specific language would provide a concrete example of the power of translation in packaging automation for effective use by developers. Ideally, students would be able to build on this experience to become more aware of the potential benefits and drawbacks of automation, more able to use it effectively in their careers, and to better understand what kind of tasks are best automated and what kind of tasks should be done by humans.

One way to address abstraction would be to develop a type system for the domain-specific language. Ideally, the students would formulate a formal connection between the semantics of the full language and the abstract semantics that the type system induces. They would understand the purposeful discarding of information involved in the abstraction and the benefits and drawbacks of discarding the information. They would be able to generalize the experience to formulate frameworks for abstraction in other fields, be able to understand the benefits and drawbacks of using abstractions, and be prepared to identify (and if appropriate discard) otherwise implicit abstractions present in a variety of fields.

5.2 Different Language Paradigm

Instead of developing a domain-specific language, one could instead start with an existing language whose basic approach and concepts differ substantially from those of conventional programming languages. In this case, going through the exercise would help students understand the importance of different perspectives and how perspectives shape the understanding and engineering process. The automation and translation aspect could be covered by studying the implementation of the logic programming language. One productive approach could be to ask the students to implement a simple language based on the core concepts of logic programming. Developing an analysis for the language, either to support efficient implementation or to check certain desirable properties, could be an effective way to introduce the concept of abstraction.

The important point here is not that logic programming is necessarily valuable in and of itself or that students will necessarily use the specific concepts in logic programming in their careers. Rather, the use of logic programming in an appropriately structured course (like the use of other non-mainstream approaches) can illustrate general concepts and make those concepts accessible to students in an immediate and concrete way.

6. Conclusion

The field of programming languages provides a uniquely favorable context for teaching certain fundamental concepts that arise in almost all modern human endeavors. These concepts include:

- Understanding the central role that one's perspective plays in the entire problem conceptualization, formulation, and solution process,
- Understanding that multiple perspectives are available, with different perspectives appropriate for different problems,
- Understanding how automation can greatly improve human productivity and that there is an appropriate division of tasks between automated systems and humans, and
- Understanding a precise notion of abstraction, that abstraction is inevitably present in every activity, and that different abstractions are appropriate for different purposes and situations.

Most people will immediately accept these concepts; some will even claim that they are obvious. But approaching these concepts through programming languages makes them precise, explicit, and clear. Students can work with them in an interactive setting that provides immediate feedback and requires completely precise thinking. This approach can therefore provide part of the solid, foundational understanding that all individuals, and not just computer scientists, need to think clearly and be truly effective in whatever endeavor they choose to pursue.