Abstract

The First Language Problem refers to the situation we find ourselves in regarding the choice of an introductory programming language. The authors argue that a modern language specifically designed to teach sound fundamental programming principles is needed but that unlike previous efforts at teaching languages, it must eventually embody all of the features of an industrial instrument. The method of achieving a language suitable to the new programmer which nevertheless not merely a toy language is achieve by a novel approach to subsetability. Other specific features needed for such a language are also discussed as well as the novel compilation issues aroused by such a language.

Keywords: Programming Languages, Concurrency in Programming Languages, Pedagogic Languages, Java Virtual Machine, compilers.

1 Introduction

It is ironic that the creative efforts in language design of the late 70's have culminated in C++ as not only a de facto industry standard, but an introductory teaching language. Teachers and textbook writers have so bravely accommodated themselves to its domination that its flaws as a working and pedagogic instrument are something of an open secret in the academic community. When we reflect on it, we are aware of the fact that, because of its C language substrate, it forces on the programmer many highly non-orthogonal features and, in a pedagogic context, obscures basic principles of imperative programming.

The consequences of using C++ as first language are compounded by the apparent commitment to teach the entire language at the introductory level. The feature richness of C++ (or Java) directly impacts programming pedagogy in a way that is rarely appreciated. Guided by the ultimate goal of making students masters of the entire language, the design problem for textbook writers is to contrive an ordering of the features that is intelligible to beginners but also ends up providing full coverage of the language. The preoccupation results in textbooks in which each chapter is cluttered with this or that quirk - (quotes within quotes, when blocks require braces, as well as oddities that are necessarily meaningless to the beginner). As a consequence, the main purposes of introductory programming instruction, namely, problem-solving and algorithmic fluency, is nowhere prominent. Command of the most important and transferable skills develops more slowly than necessary. Instead of understanding programming as the composition of computational primitives to some purpose, students identify programming with the command of a large and bewildering repository of features. The faddish drive to use Java for introductory purposes confirms this perception: few argue directly that Java is well-suited as a teaching language; only that it is better than C++.

The reluctance to adopt a specialized teaching language has causes that go beyond the historical influence of Unix and C. The sheer and ever growing size of the computer science curriculum and the responsibility of rendering graduates capable in more and more areas of the discipline makes the adoption of a teaching language seem a luxurious waste. The design of computer science curricula are driven by the implicit principle that it should include nothing that is not of permanent worth, i.e., nothing that will not reappear in later courses or experience.

2 Ideal Properties of a First Algorithmic Language

In spite of the realities reviewed above a specialized first language is not impossible, but it does impose design constraints that, for example, were irrelevant to Wirth when he designed Pascal [5]. The reluctance to waste curricular time on a toy instrument
means that a teaching language must also be industrial strength and must strongly resemble in look and feel one that will later be put to production use. We believe that this requirement can be met. A teaching language need not be weak; it just has to include teachability as a design constraint, in the same way that hardware sometimes must include testability as a design constraint. What follows are our reflections on the implications of language design under the constraint of teachability.

2.1 Lossless Segue

To accommodate the reluctance to introduce syntax solely for pedagogic purposes, a teaching language must be strongly convergent with a real-world instrument in common use, it must have the look and feel of Java or C++ or some other langage du jour.[6,7]

There is historical precedent to support this constraint. Wirth introduced Pascal specifically to be used as a pedagogic tool to teach a rigorous approach to what was then referred to as structured programming. The language was so well constructed that it soon achieved a following outside the academic community and was widely utilized in industry.

Pascal began to be supplanted because it did not directly address real-world necessities such as separate compilation and, for a while, it returned to its previous status as a teaching language. [8,9,10] Since, as we believe, mere teaching languages are unwelcome to the academic community, any language wishing for that status must segue almost seamlessly with one used in industry and advanced parts of the curriculum.

2.2 Compilability

The above constraint implies some discipline in the feature set of a language. If a teaching language must segue easily to some industry standard, it must not introduce powerful constructs that require interpretation for their computational realization. [11,12,13] Thus, an overriding constraint in the design of the language must be that the features selected be capable of being realized in highly compiled fashion.

2.3 Constructs for Concurrency

The computing systems students will eventually be tasked to develop will often be multithreaded. The exigencies of GUI programming demand not only an object-oriented perspective but also a multi-threaded one. Given this reality it is nothing short of amazing to observe the way that the concepts of concurrent programming are dealt with in the standard undergraduate computer science curriculum. The traditional computer science curriculum is mute on the issue of concurrency until the senior year operating system course. This material was relegated to this course in times long ago, as it was necessary to achieving an understanding of how the operating system was built, an operating system itself being the quintessential example of a concurrent program. The days when the important area of concurrent programming could be shelved until the senior year have long since passed, yet the typical undergraduate computer science curriculum fails to address this. This flaw in the undergraduate curriculum manifests itself in many ways. Its influence on the primary issue at hand is that we continue to teach students fundamental courses in programming, data structures and algorithms without so much as the slightest mention of concurrency. The languages the students utilize to formulate solutions to problems are totally sequential while the environment they will be dealing with upon graduation is totally multi-threaded. Industry is no better off in this regard than academia. Application software for multi-thread GUI environments is developed in C++, a language totally devoid of any concurrency features. Multi-threading and synchronization are dealt with via the invocation of methods in class libraries. This is the wrong abstraction! Concurrency should be dealt with through high-level language features not calls to library routines. In this respect we see that C++ is not only the wrong tool for academia it is also the wrong tool for industry.

2.4 Separate Algorithmics from the Machine

All serious programmers must learn something about the behavior of the computers on which their code executes and the mapping of high-level constructs to the underlying machine. But this necessary knowledge is not to be confounded with fluency in algorithm development. The introduction of pointers and memory allocation, the performance implications of parameter passing modes do not introduce new algorithmic models; rather, they introduce new primitives. A good teaching language will support the use of powerful data structures without forcing a prerequisite knowledge of their realization as pointer based structures.

2.5 Powerful, Enduring Data types

Although algorithmic programming is a computational realization of the logic of imperatives, certain constructions having little sense outside programming have a special place within it, such as stacks, lists, queues and (hash) tables. Most programming languages force a conceptually irrelevant prerequisite upon the use of such structures, namely, pointers and memory allocation. It is important to note
that understanding the use of such structures to build powerful programs does not depend upon facility with pointers. Pointers and memory allocation concern a program’s relation to the underlying machine; data types with insertion and access disciplines concern algorithmics. An ideal teaching language will make organizing programs around finite state or pushdown automata straightforward by offering up appropriate data types as part of the language.

2.6 Avoid Syntactic Irrelevancies

A second reason that stacks, lists, queues and tables should be built-in data types is that the only alternative is force students explicitly to import them from libraries. This would exacerbate the problem already aroused by C++ and Java of having students blindly type in swaths of programming text whose meaning must initially be unexplained and the requirement for which must be a matter of faith. (“Pay no attention to the man behind the curtain!”).

2.7 Subetability

The most important feature of a teaching language is subsetability. All teachers of elementary programming sometimes wish that students be temporarily denied certain parts of a language, both to prevent the floundering that occurs when students roam into areas of the language they are not prepared to understand (sometimes on the advice of other misinformed peers) but also simply to serve as an implicit guide to the relationship between particular configurations of constructs and particular problem situations. For example, the most important skills in imperative programming can be taught and honed with a language that provides no more than basic control structures, integer types and I/O (including for convenience literal string output). With just these features, it is straightforward to devise problems from, for example, number theory that force the construction of all possible permutations, combinations and nestings of control structures up to a reasonable level of complexity. Note that the requirement of maximal subsetability does not dictate a concentric family of languages whose feature-set is predefined by the language designer (in the spirit of the PL\textsubscript{\textsc{s}} series of languages). Rather, we propose that language subsets be selected by the user of the language in any way that is syntactically and semantically feasible. In this way, the language would support a variety of curricular philosophies. Ideally, a teacher would be able to construct subset languages that make little sense for a production language. For example, for a particular problem, to guide students to a particular algorithmic realization.

Maximal subsetability is not just a desirable feature of a teaching language. For industry coding standards often take the form of restrictions on the feature set of a language. The ability to “turn-off” a feature

3 Language Design and Processing Research Issues

The requirements that an algorithmic first language be capable of being subseted in extreme ways and that it be highly compilable with high level intrinsic data structures arouses several interesting research issues concerning language design and compilation.

3.1 Cafeteria Style Language Assembly

The requirement of maximum subsetability entails that a teacher or other user should be able to combine the primitives of language in any way that does not violate syntactic and semantic dependencies. This requirement entails that the grammatical formalization of the language systematically honors these syntactic and semantic dependencies. It is not at all obvious that this is possible. The ability to define a language consisting of an unconstrained selection of a subset of features defined in the full realization of the language poses interesting theoretical issues. This requirement also implies that the implementation of the compiler for the language be carried out in such a way as to facilitate the automated generation of an appropriate compiler for a subset of features selected. This, in turn, requires that implementation details not contained within the grammar specification be denoted in a way that facilitates automated construction of the semantic analyzer and code generator. These requirements create demands that are on the very edge of current compiler construction techniques.

3.2 Orthogonality, Powerful Data Types and Compilability.

The requirement that a teaching language include as pre-defined types stacks, lists, queues and tables together with the natural requirements of language design, entails that these structures permit—unless eliminated by a subsetting operation—any nesting and combination of such features. This arouses unprecedented difficulties for the compiler. We wish to retain the highly compiled nature of the language and not resort to excessive degrees of interpretation at execution time yet also provide these high-level intrinsic data structures as programming language primitives. Similar powerful intrinsic data structure capabilities are normally associated with languages that utilize a high degree of run-time interpretation such as LISP, ICON or SNOBOL.
4 CONCLUSIONS

We have discussed properties of an ideal teaching language which are very much at odds with languages in common use in industry. Given the reluctance to devote curricular time to toy languages, we propose a teaching language which in its fullest realization can serve as a production language but which can be subseted to suit a particular teaching philosophy. Endorsement of this view also implies the presence of powerful intrinsic data types. These two major teaching requirements present the language designer and compiler writer with novel research problems.

5 REFERENCES


ISBN 0-201-40353-6