Seven Deadly Sins of Introductory Programming Language Design

Linda McIver & Damian Conway
Department of Computer Science
Monash University, Victoria, Australia

Abstract

We discuss seven undesirable features common to many programming languages used to teach first-time programmers, and illustrate typical pedagogical difficulties which stem from them with examples drawn from the programming languages ABC, Ada, C, C++, Eiffel, Haskell, LISP, Modula 3, Pascal, Prolog, Scheme, and Turing. We propose seven language design (or selection) principles which may reduce the incidence of such undesirable features.

Introduction

Learning to program is difficult. We believe that a substantial part of this difficulty arises from the structure, syntax and semantics of the programming languages which are commonly used to teach programming.

Programming language designers are (of necessity) highly intelligent experts in the field of programming, and are consequently far removed both temporally and cognitively from the difficulties experienced by the novice programmer. This gulf of experience and ability results in languages which are either too restrictive or too powerful (or sometimes, paradoxically, both).

We divide introductory programming languages into two broad categories: special purpose teaching languages (such as Pascal [1], Turing [2], ABC [3]) and popular "real-world" languages (such as C [4], C++ [5], Ada [6], Modula 3 [7], Haskell [8], and Scheme [9]).

There is a long history of scholarly and less-than-scholarly debate [10,11,12,13] regarding the comparative merits and flaws of many of these languages. Typically this debate centres on theoretical issues (such as expressiveness, range of concepts supported, or paradigmatic integrity) and practical considerations (such as range of available platforms, support tools and environments, or efficiency). This paper departs from that tradition in that it focuses exclusively on the issues which arise in the context of teaching introductory programming.

We enumerate seven serious pedagogical problems, each of which is common to most or all of the above-mentioned languages, even those specifically designed for teaching. We also propose a like number of design principles, which we believe will lead to the development of significantly more "teachable" introductory programming languages. Finally we suggest seven criteria which educators may find useful in evaluating existing languages for introductory teaching.

The observations and suggestions contained in this paper have been developed as part of the GRAIL[1] project within the Department of Computer Science, Monash University. The aim of this project is to study the early acquisition of programming skills, with the ultimate goal of creating more usable languages and support tools.

Seven Deadly Sins

1. Less is more.

The "less is more" principle appears in many forms, almost all of which seem to be ultimately detrimental to the learning process. Perhaps the most obvious examples are the Scheme language and other LISP variants. Scheme has effectively only one data type – the list – and one operation – evaluation of a list. While this abstraction is very simple to explain, and not difficult for the beginner to grasp superficially, it does result in code that is difficult to read because of large numbers of nested parentheses and the absence of other structuring punctuation.

Furthermore, to support this extreme degree of homogeneity, a large number of inbuilt functions are required, many of which are quite sophisticated in their behaviour, and therefore difficult to understand and use correctly (for example: `sort` *vs* `sortcar` in Franz LISP [14]).

A "less is more" approach is usually justified in terms of paradigmatic purity: strict adherence to a single functional, logical or object oriented paradigm. While this orthodoxy can make for a certain conceptual simplicity and elegance (which can be of considerable advantage in teaching concepts such as scoping, recursion and encapsulation), in practice it can also lead to extremely obscure and unreadable code. In some cases, relatively simple programs must be substantially restructured to achieve even basic effects such as input and output.

The underlying pedagogical difficulty is that students are not used to solving problems in a single pure paradigm. Much of the problem solving they do in the real world is procedural in nature (cooking a meal, totalling a restaurant bill, etc.), but other problems with which they are familiar are more amenable to constraint solving (dispute resolution, holiday planning, budgeting), a functional or pipe-line approach (collaborative tasks, various types of component assembly), or even object-oriented methods (using an automatic teller machine, learning physical skills).

The results of enforcing paradigmatic purity can be as simple as the annoying requirement in Turing that functions have no side-effects, or as far reaching as the mysteries of

[1] Genuinely Readable And Intuitive Languages

I/O in Haskell, which sometimes necessitate the warping of the entire structure of an otherwise elegant and comprehensible functional program.

2. More is more.

It is equally true that many languages are based on design philosophies which err in the other extreme. Powerful, real world languages (C++ and Ada, for example) are amongst the prime culprits here. Often such languages are taught by subsetting – teaching a small but usable part of the language whilst ignoring its more powerful features.

At first glance this approach seems quite reasonable, but two pedagogical problems frequently sabotage it. The first is that textbooks and other reference materials rarely confine themselves to the selected subset. The second is that, even if the textbook does limit itself to the required subset, the compiler almost certainly does not. The result is often worse than if the complete language was taught: students must still contend with the full semantics of the language, but much of it has deliberately not been explained to them!

C++ is certainly one of the most popular languages in "real-world" use and (for that very reason) is also increasingly widely taught as an introductory language. One of the justifications typically cited for teaching C++ [11] is the range of low- and high-level features it provides (from bit manipulation of raw pointers, to templated abstract classes with polymorphic member functions).

Beginners, however, are notoriously poor at maintaining two or more conceptual perspectives simultaneously [15]. Dichotomies of perspective (such as syntax *vs* semantics, static *vs* dynamic structure, process *vs* data) complicate the teaching of any programming language. The availability of very low-level implementation-oriented constructs and high-level solution-oriented features in a single language only serves to increase substantially the already considerable cognitive demands placed on the student.

As well as the obvious concerns regarding learning curves, confusion of levels, and the difficulties of adequate error detection, a wide range of features necessitates a commensurately complex syntax and often also entails a host of implicit operations and function calls, automatic conversions, type inferences, and resolutions of overloaded functions, variable and function scoping.

Examples of this "creeping featuritis" are easy to cite: C++ provides over 50 distinct operators at 17 levels of precedence, Ada9X has 68 reserved words and over 50 predefined attributes, Modula 3 reserves over 100 keywords, and some commonly-used LISP dialects ([14] for example) define over 500 special functions. Because most textbooks and compilers attempt to cover the full language, novice programmers are forced to contend with all of these features, even if they are not using them.

3. Grammatical traps.

Another class of pedagogical problems stems from various kinds of confusing syntactic and semantic constructs which are present in most introductory languages. Some of these constructs arise from the constraints of the ASCII character set, whilst others are the result of a deliberate but (in our view) misguided "less-is-more" design policy. The common feature of these problems is that they are analogous to certain sophisticated grammatical constructs in natural languages, and result in the same types of learning problems as are seen in natural language acquisition.

One such construct is the *syntactic synonym*, in which two or more syntaxes are available to specify a single construct. An common example of this is dynamic array access in C, wherein the second[2] element of an array may be accessed by any of the following syntaxes, some of which are legal only in certain contexts:

```
array[1]   *(array+1)   1[array]   *++array
```

A less well-known example is list construction. In Haskell the list construction expression `[1,2,3]` is synonymous with `1:(2:(3:[]))`. In Prolog `[1,2,3]` is equivalent to `.(1,.(2,.(3,[])))` and both produce a third form on evaluation: `(1,2,3)`.

In themselves, synonyms are minor irritants (which multiply the learning curve for particular constructs by no more than 200–300%) However, they can also have a more serious and insidious effect by blurring the underlying programming concept in the student's mind, because that concept is no longer associated with a single clear syntax.

Syntactic homonyms (constructs which are syntactically the same, but which have two or more different semantics depending on context) are perhaps a more serious flaw in a language. An extreme example of this[3] may be seen in the pedagogically-oriented language Turing, in which the construct `A(B)` has five distinct meanings:

- call function **A** and pass parameter **B**
- dereference the pointer **B** to access an object in collection **A**
- access the **B**th element of array **A**
- construct a set of type **A** with a single element having the value of **B**
- create a one-letter substring of the string **A** consisting of its **B**th character

The student, armed with only a fuzzy understanding of the differences between these concepts, finds no support from the syntax. It should be noted that the decision to overload this construct was taken quite deliberately and on pedagogical grounds:

> "Notice that referencing an element of array **a** with subscript **i** as in **a(i)** is notationally equivalent to [the pointer dereference] **c(p)**. This is an example of *uniform referents,* which means that analogous ways of accessing data should be notationally equivalent." [17]

Another difficult grammatical construct which frequently appears in languages is *elision* (the omission of a syntactic component). C is well known for its default integer return value and its curious string literal concatenation

2 The fact that **array[1]** refers to the second element of array is itself a grammatical trap.

3 But not as extreme as LISP and its variants, which could be viewed as one massive homonym.

behaviour, but default behaviours occur in many languages: automated sorting of lists in ABC, type inference in Haskell and Turing, LISP superbrackets, switch fall-through in C++, etc.

4. Hardware dependence.

In addition to battling the various syntactic and semantic levels of an introductory language, the novice programmer is often forced to contend simultaneously with the constraints of the underlying hardware (sometimes merely for the convenience of the compiler writer).

This "closeness to the metal" is particularly noticeable in the design and implementation of basic numerical and character string types. There seems no convincing reason why the novice student, already struggling to master the syntax and semantics of various constructs, should also be forced to deal with the details of representational precision, varying machine word sizes, awkward memory models, or a profusion of conceptually-equivalent but semantically-distinct data types.

The semantics of arrays in Pascal, in which the novice must grapple with the fundamental type difference of arrays of different lengths, is a notable example. The presence of thirty-two distinct numerical data types in C/C++[4] is another. These types are particularly problematical in C as they are generally not portable. The standard **`int`** type, for example, varies from 16 to 32 bit representations depending on the machine and the implementation. This can lead to strange and unexpected errors when overflow occurs. A student whose program attempts to add a $4,000 bonus to a $30,000 salary may be justifiably confused to find that the result is a negative number.

5. Backwards Compatibility.

Backwards compatibility is a useful property from the experienced programmer's point of view, as it promotes reuse of both code and programming skills. The novice however can take no advantage of these benefits and must instead bear the pedagogical costs they entail.

Backwards compatibility comes in two major forms: "genetic" and "memetic". Whilst both forms can lead to pedagogically suspect decisions during the design of a language, genetic compatibility is generally the result of a conscious decision on the part of the language designers, whereas memetic compatibility is frequently inadvertent.

Genetic compatibility is exemplified by the relationship between languages such as C++ and C [16], Scheme and LISP [9], and Turing, Euclid and Pascal[17] , and results from the decision to retain the semantics and often the general syntactic "look-and-feel" of an ancestor language. Genetic compatibility need not of course imply the near complete backwards compatibility as seen in the C/C++ relationship (Turing and Scheme differ significantly from their ancestors), nonetheless languages which attempt a significant degree of historical consistency inevitably perpetuate some problematical constructs.

Language designers occasionally acknowledge the problems that their quest for genetic compatibility produces:

> "At this point, the reader may be confused at having so many ways to define a [Haskell] function! The decision to provide these mechanisms partly reflects historical conventions, and partly reflects the desire for consistency (for example, in the treatment of infix vs. regular functions)." [8]

> "Over the years, C++'s greatest strength and its greatest weakness has been its C compatibility. This came as no surprise." [16]

As well as introducing (or compounding) the problems inherent in a "more-is-more" approach, the addition of new concepts to an old language often leads to inconsistency of abstraction (consider the differing semantics of **`TEXT`** and other array types in Modula 3), the creation of synonyms or homonyms (array indexing, function calls and pointer dereference in Turing), as well as the perpetuation of outmoded or flawed constructs (such as **`char*`** strings in C++) or syntax (for example, the inexplicably-named[5] **`car`** and **`cdr`** which Scheme inherits from LISP).

Not all syntactic or semantic inheritance stems from deliberate decisions regarding backwards compatibility. Some constructs and symbols seem to propagate memetically across language family boundaries, and have become de facto standards within the programming community. This is often despite the fact that such constructs may have been viewed by their progenitors as ad hoc and may indeed have no discernible connection with the concepts they are intended to represent. Memetic compatibility is surprisingly pervasive and may be seen in the widespread use of "standard" symbols such as **`*`** for multiplication, **`=`** or **`:=`** for assignment, **`array[index]`** for indexing.

The major pedagogical problem with the presence of such syntactic memes is that they significantly reduce the degree to which the novice, an outsider to the programmer culture, can rely on existing knowledge (such as $\times$ meaning multiply, or a subscript representing an index).

Unfortunately, memetic compatibilities can also be particularly difficult to identify (and their pedagogical effects correspondingly hard to analyse), precisely because both the language designer and the programming teacher are so familiar with them.

6. Excessive Cleverness.

Instances of "excessive" cleverness can be difficult to spot, precisely because the "excess" exists only relative to the knowledge level of the novice. Frequently the only way to detect excessive cleverness is to see a novice programmer's complete misunderstanding of an "obvious" concept.

---

4 **`int`**, **`short`**, **`long`**, **`unsigned int`**, **`unsigned short`**, **`unsigned long`**, **`float`** and **`double`**; plus three **`const`** and/or **`volatile`** variants of each.

5 "Inexplicable" in the sense that explaining that they derive from "contents of address register" and "contents of decrement register" respectively, rarely assists the student's comprehension or recall.

The premier example of the adverse effects of cleverness in programming language design (and one which is obvious to programmers at all skill levels) must surely be the C/C++ declaration syntax [10]. On the surface, it seems like an excellent notion: let declarations mirror usage. Unfortunately, the very concept undermines the principle of visually differentiating semantic differences. Students have enough trouble mentally separating the concepts of declaration and usage, without the syntax conspiring to blur that crucial difference.

Other examples of detrimental cleverness are less obvious, but still easily come by. For example, some languages (ABC, Haskell and Python, for instance) use indentation to specify scope. This eliminates the need for grouping constructs (such as bracketing or **BEGIN**...**END** pairs) but fails to take into account the apparently inability of many students to master the concept and practice of consistent indenting. Indentation-based scoping also eliminates the useful redundancy of employing both syntax (delimiters) and convention (indenting) to reinforce semantics.

Sometimes a genuinely clever idea can be sabotaged by its own syntax. For example, in Turing dynamic memory may be partitioned into strictly typed "collections" which are then capable of storing dynamically allocated instances of a single data type. Pointer variables may be associated with a particular collection and can only be used to refer to data within that collection. This approach provides strong type checking of dynamic memory and enables the compiler to catch and accurately diagnose the majority of common pointer manipulation errors.

Unfortunately, this genuinely clever idea is disastrously undermined by poor choice of syntax:

```
%DECLARE A COLLECTION STORING SomeDataType
    var collectionName:
                collection of SomeDataType

%DECLARE A POINTER
    var ptr : pointer to collectionName

%INSTANTIATE A NEW OBJECT OF SomeDataType
    new collectionName, instance
```

Students immediately (but erroneously) conclude that:

- **collectionName** is a variable (it's actually a reference to a partition of dynamic memory and does not have the full semantics of a variable.)
- **ptr** can be used to point to **collectionName** (it can only be used to point at instances of **SomeDataType** allocated within the partition accessed via **collectionName**.)
- **instance** is a new instance of type **collectionName** (**instance** points to an instance of type **SomeDataType** newly allocated within **collectionName**.)

7. Violation of Expectations.

As the last example in the previous section indicates, violating a reasonable expectation is probably the worst pedagogical sin that an introductory programming language designer can commit. Some examples are very well-known, such as the almost maliciously designed C/C++:

```
if (x=0 || -10<y<10)  { /* WHATEVER */ }
```

in which the condition *always* evaluates to true (regardless of the values of **x** and **y**) whilst the value of **x** is silently reset to one. Particularly insidious is the fact that this code is perfectly legal and compiles without even generating a warning under many compilers.

A less obvious example of syntax violating expectations is the use of **%** as a comment introducer in Turing. The following code is syntactically correct and semantically valid, but will result in unexpectedly low pass rates:

```
passMark := maxMark * 50%
```

Semantic violations of expectation are even less excusable, but regrettably more common. For example, consider the list type in the ABC programming language. A novice, having written a seemly straight-forward program to store a list and then print the first element:

```
PUT {"first"; "third"; "fifth"} IN list
WRITE list item 1
```

may well be considerably puzzled and disheartened when the program prints: **"fifth"**

The blame for this minor failure can hardly be laid on the novice, who may simply have forgotten (or perhaps never grasped) that ABC lists are automatically sorted on input. The fault lies squarely with the language designers, for although a sorting function is an extremely useful capacity in a language, hidden side effects such as this can be highly confusing for the inexperienced user, especially when the "magic" gets in the way of the programming task.

Even the semantics of fundamental and nearly universal programming memes, such the **while** loop and the finite precision integer, can be surprisingly difficult for students to comprehend. A **while** loop doesn't execute "while" its condition is true, but rather until its condition ceases to be true at the end of its associated code block.

Finite precision integers don't obey the familiar rules of whole number arithmetic and can also cause much confusion when overflow, underflow or truncation produce consequential errors (which may manifest well after the actual numerical error occurred).

Prolog[6] offers a programming system based on predicate calculus, but with silent and deadly caveats. The novice attempting to create a recursive definition (of a list membership predicate, for example) will quite reasonably construct something like:

```
member(X,[_|Y]) :- member(X,Y).
member(X,[X|_]).
```

which is logically quite correct, but which always fails because the declaration order of predicates determines the sequence of predicate unification in Prolog.

The Prolog equality operator (**X=Y**) violates expectations in another way, in that it implies an assignment of reference as a side effect:

[6] Although Prolog is rarely used as an introductory programming language, many more advanced students eventually find themselves in the role of Prolog novice.

"If **X** is an uninstantiated variable and if **Y** is any object, then **X** and **Y** are equal. As a side-effect, **X** will be instantiated to whatever **Y** is. [When **X** and **Y** are both uninstantiated,] the goal succeeds, and the two variables share. If two variables share, then whenever one of them becomes instantiated to some term, the other one automatically is instantiated to the same term." [18]

Seven Steps Towards More "Teachable" Languages

1. Start where the novice is.

A fundamental aspect of learning is the process of assimilating new concepts into an existing cognitive structure [19,20,21]. This process, known variously as *connecting*, *accretion* or *subsumption*, is made all the more difficult if parts of the existing structure have to be removed (*unlearning*) or restricted (*exceptions*). Hence, the novice who must unlearn that × or • means multiply, and then substitute ***** in a programming context, faces a harder learning task than the student who can continue to put their knowledge of × to use. Similarly, students have a large corpus of knowledge regarding integer and real arithmetic, which cannot be capitalised upon if they must disregard it to cope with finite precision representations.

Another example of this type of difficulty is the use of **=** variants for assignment. Many students, when confronted with this operator, become confused as to the nature of assignment and its relationship to equality. For example, seeing the following sequence of statements :

```
X = 3;        Y = X;        Y = 7;
```

novice students sometimes expect the value of **X** to be equal to **7** (since "Y and X are equal"). The equivalent sequence:

```
X ← 3;        Y ← X;        Y ← 7;
```

seems to evoke less confusion, possibly because the syntax reinforces the notion of procedural *transfer* of value, rather than transitive *equality* of value.

We have shown over one thousand novice programming students the C/C++ expression:

```
"the quick brown fox" + "jumps  over a lazy dog"
```

and asked them what they believe the effect of the **+** sign is. Not one of them has ever suggested that the **+** sign is illegally attempting to add the address of the locations of the first characters of the two literal strings. Without exception, they believed that the **+** should concatenate the two strings.

We believe that introductory languages should be designed so that reasonable assumptions based on prior non-programming-based knowledge remain reasonable assumptions in the programming domain. In other words, the constructs of a teaching language should not violate student expectations. Note that this principle has both syntactic and semantic implications in the selection and definition of operators, functions and inbuilt data types.

2. Differentiate semantics with syntax.

Novices experience great difficulty in building clear and well-defined mental models of the various components of a programming language. Syntactic cues can be of significant assistance in differentiating the semantics of different constructs.

Constructs which may, to the accomplished programmer, seem naturally similar or analogous in concept, functionality, or implementation (for example: using the integer subset {0,1} as a substitute for a true boolean type, arrays being analogous to discrete functions of finite domain and range, arrays being implemented via pointers) still need to be clearly syntactically differentiated for the novice.

We believe it is misguided to highlight the similarities between such constructs with similar (or worse, identical) syntaxes, because it initially blurs the crucial differences by which students can first discriminate between programming concepts, and later robs them of the opportunity to consolidate understanding by identifying these underlying conceptual connections themselves.

3. Make the syntax readable and consistent.

Novice programmers, like all novices, have a weak grasp of the "signal" of a new concept and are particularly susceptible to noise. This suggests that an introductory programming language should aim to boost the conceptual signal and reduce the syntactic noise. One obvious means of improving the S/N ratio is to choose a signal with which the recipient is already familiar. For example: **if** rather than **cond**, **head/tail** rather than **car/cdr**, × rather than *****.

Another approach is to select signals which are consistent, distinct, and predictable. For example, delineating code blocks within constructs by **<name>**...**end<name>** pairs:

```
loop
    if isValid(name)
        exit loop
    end if
    output name
    name ← getNextName()
end loop
```

It can be difficult to steer an appropriate path between the syntactic extremities of "less-is-more" and "more-is-more". On one hand, reducing syntactic noise might involve minimizing the overall syntax, for example:

```
if  x ≠ y
    y ← x
else
    x ← y
```

rather than

```
if (x <> y) {
    y:=x;
}else{
    x:=y;
}
```

Alternatively, it may be better to *increase* the complexity of the syntax in order to reduce homonyms which blur the signal. For example, the meaning of the various components of the Turing expression[7]:

```
f(C(p).A(I))(N)
```

might be better conveyed with the syntax:

f(C::p->A_I)[N]

The second form, whilst regrettably no more mnemonic than the first, does at least provide adequate visual differentiation between pointer dereference, array indexing, function call, and substring extraction.

4. Provide a small and orthogonal set of features.

Homonyms and synonyms are an acute problem in the design of a teaching language. We believe the best way to minimize these pedagogical impediments is to select a small set of non-overlapping language features and assign them distinct (and mnemonic) syntactic representations.

A side effect of this recommendation is that, as the number of language constructs is restricted, those which *are* chosen must inevitably become more general and probably more powerful. In particular, we believe that it is important to provide basic data types at a high level of abstraction, with semantics which mirror as closely as possible the "real-world" concepts those data types represent.

For example, we would suggest that an introductory language should not provide separate data types for a single character and a character string. Rather, there should be a single "variable length string" type, with individual characters being represented by strings containing a single letter. A full complement of string operators should be supplied, with operators for assignment, concatenation, substring extraction, comparison, and input/output. In addition, a set of suitable inbuilt functions (or predicates or methods, according to the language's paradigm) should be provided to implement other common operations for which operators may be inappropriate (for example: length of string, case transformations, substring membership, etc). As character strings may be strictly ordered, the string type should be a valid indexing type for case statements and user defined arrays.

Likewise, we suggest that an introductory language need only provide a single numeric data type which stores rational numbers with arbitrary precision integer numerators and denominators. The restriction to rationals still allows the educator to discuss the general issue of representational limitations (such as the necessary approximation of common transfinite numbers such as π and e), but eliminates several large classes of common student error which are caused by misapplication of prior mathematical understanding to fixed precision arithmetic. A single arbitrary precision numeric type has the additional benefit of eliminating many hardware dependence problems.

Other features which might be provided include:

- A single non-terminating loop construct, possibly modelled on the Turing or Eiffel **`loop`** statement, with an associated **`exit loop`** command which may be controlled by **`if`** statements within the loop.
- A single generic list meta-type, allowing the user to define homogeneous or heterogeneous lists, indexed by any well-ordered type (numeric, boolean, string).
- A single, consistent model and syntax for I/O.

5. Be especially careful with I/O.

With growing awareness of the importance of software usability, it is natural that students should be encouraged to engineer the input and output of their programs carefully. Too often, however, they are hampered by "more-is-more" programming language I/O mechanisms which are needlessly ornate or complicated.

The essence of I/O is very simple: send a suitable representation of a value to a device. The complexity frequently observed in the I/O mechanisms of introductory languages often stems from a desire to provide too much control over the value conversion process.

Somewhat surprisingly, the C++ language, not otherwise known for its friendliness towards the novice[8], provides a reasonable (if over-featured) model of I/O. Turing also offers a very straightforward I/O model and syntax.

We believe that the I/O mechanism for an introductory language should be defined at the same high level of abstraction as the other language constructs. We see the basic features of a good pedagogical I/O model as being:

- a simple character stream I/O abstraction, with specific streams (for screen, keyboard, and files) represented by variables of special inbuilt types.
- uniform input and output syntaxes for all data types (for example, infix "input" and "output" operators which may be applied between a stream object and a heterogeneous list of values and/or references)
- a default idempotent[9] I/O format for all data types (including character strings and user defined types), with appropriate formatting defaults for justification, output field width, numerical precision, etc.
- a reasonable, automatically-deduced output format for user-defined data types (for example, output each globally accessible data member of a user-defined ADT, one value per line)
- a simple and explicit syntax for specifying non-default output formatting (for example: a generic **`leftjustify`** function to convert any value to a left-justified character string of specified field width.)

6. Provide better error diagnosis.

There is a widely cherished belief amongst educators that one of the ways students learn best is by making their own mistakes. What is often neglected is that this mode of

[7] "Create a substring consisting of the **N**th letter of the string returned by the function **f** when passed the **I**th element of the array member **A** of the object within collection **C** which is pointed to by **p**".

[8] or indeed towards the expert!

9 Idempotence of I/O means that outputting the value of a variable and then reading that output value back into the same variable has no discernable overall effect. String I/O is non-idempotent in most programming languages, because strings are typically written out in their full length, but read in word-by-word (ie: to the first white-space character).

learning is only effective if a student's otherwise random walk through the problem space can be guided by prompt, accurate, and comprehensible feedback on their errors.

Making and correcting an error is certainly a useful experience for expert and beginner alike, but the process of correction can be tortuous without meaningful guidance. Compiler error messages are often couched in unnecessarily terse and technical jargon which serves only to confuse and distress the student. By the time the messages have been deciphered and explained by a teacher or tutor, any useful feedback which may have been gained has been largely negated by the delay and stress involved.

The type of feedback that students receive when compiling their programs typically runs along the lines of:

```
Syntax error near line 4
Not implemented: & of =
No subexpression in procedure call
Application of non-procedure "proc"
```

Even should they manage to compile their program, run time errors typically produce useful feedback like:

```
Segmentation violation: core dumped
Application "unknown" exited (error code 1)
<<function>>
```

Error diagnosis is a weak point of most compiler technology, yet it is this compiler feature that novices spend most of their time interacting with. Whilst well-designed error feedback is not unknown (Turing is exemplary in this respect) many language implementations, particularly interpreters, have little or no error diagnosis. In these cases, errors are detected when the program executes in some unexpected way. Detecting and correcting errors in these implementations can be extremely difficult, particularly for a beginner, who may be uncertain what the expected behaviour of the program actually was.

For an introductory language, error messages should be issued in plain language, not technical jargon. They should reflect the syntactic or semantic error that was discovered, rather than the behaviour of the parser. Error diagnosis must be highly reliable or, where this is infeasible, error messages must be suitably non-committal. For example, given the statement:

```
int F(X);   // WHERE X IS AN UNDECLARED TYPE
```

a widely-used C++ compiler emits the error message:

```
')' expected
```

rather than explaining that:

```
An unknown type 'X' was used in the
parameter list of function 'F'.
```

In this case even a vague message like:

```
There seems to be a problem with the
parameter list of function 'F'.
```

would be of more use.

We suggest that a fully-specified error reporting mechanism should be an integral part of any introductory programming language definition. Such a mechanism must mandate plain language error messages and should ideally provide multiple levels of detail in error messages (possibly through a "tell-me-more" option).

Common compilation errors (such as omitted end-of-statement markers or mismatched brackets) should be accurately diagnosed and clearly reported. Cases where the root cause of an error is not easily established should be reported as problems of uncertain origin, with one or more suggested causes offered in suitably non-committal language.

Run-time errors should likewise be clearly and accurately reported, at the highest possible level of abstraction. It is sufficient for the expert to be informed that a segmentation fault has occurred, but the novice needs a hint as to whether the event was caused by an array bounds violation, an invalid pointer dereference, an allocation failure, or something else entirely.

7. Choose a suitable level of abstraction.

When first introduced to programming, students often have trouble finding the correct level of abstraction for writing algorithms. Some expect a very high level of understanding from the computer, to the extent of assuming that variable names will affect the semantics of the program (for example, believing that naming a function `max` is sufficient to ensure that it computes the maximum value of its arguments). Others attempt to code everything, including basic control structures, from scratch. To require algorithms to be coded in languages with extreme levels of abstraction (for example: high-end logic, functional or pure object-oriented languages, or low-level assembler) merely compounds the students' already abundant confusion.

It is critical for an introductory language to approximate closely the abstraction level of the problem domain in which beginners typically find themselves working. Hence it is appropriate to provide language constructs suitable for dealing with basic numerical computing, data storage and retrieval, sorting and searching, etc. For most introductory courses, language features which support very low-level programming (for example: direct bit-manipulation of memory) or very high-level techniques (such as continuations) will merely serve to stretch the syntax and semantics of the language beyond the novice's grasp.

Seven Criteria for Choosing an Introductory Language

Each of the preceding design principles also provides a criterion against which to evaluate the suitability of existing programming languages for introductory teaching. We would suggest that when evaluating a potential teaching language, in addition to addressing the usual considerations (such as language paradigm, compiler availability, textbook quality, establishment and maintenance costs, popularity, etc.), educators should also bear in mind the seven "sins" we have enumerated. The key questions are:

- Is the syntax of the language excessively complex (and therefore lexically "noisy") or too sparse (and therefore insufficiently discriminating)? Are there

syntactic homonyms, synonyms or elisions which may confuse the novice?
- Are the control structures, operators and inbuilt functions of the language reasonably mnemonic? Are they likely to be consistent with students' previous (mathematical) experience? How much "unlearning" will they require of the novice?
- Are the semantics of the language inconsistent, obscure, or unnecessarily complicated? Are there constructs whose invocation, behaviour or side-effects are likely to confuse a student or violate novices' reasonable expectations?
- Are the error diagnostics clear and meaningful at a novice's level of understanding? Are they unambiguous, detailed and not overly technical? Are they accurate where possible and non-committal otherwise?
- Are parts of the language subject to unnecessary hardware dependencies or implementation-related constraints? Will necessary restrictions be difficult to explain to the novice?
- Is the language too big (over-featured) or too small (restrictive)? Is the level of abstraction of the language constructs appropriate for the practical components of the course?
- Are the apparent virtues of the language equally "apparent" from the novice's perspective? Alternatively, are they a product of the educator's familiarity with the candidate language or with the languages which were the candidate's genetic and memetic influences?

In the real world it is clear that the choice of any one language must necessarily be a compromise between economic, political and pedagogical factors. The relative importance of each of these considerations will depend on the specific aims and priorities of the institution, educator and course. Unfortunately, all too often pedagogical factors are neglected, or sacrificed to more obvious and prosaic concerns. We believe that this approach undermines the ultimate goal of successful student learning.

Conclusion

We have enumerated seven ways in which introductory programming languages conspire to hinder the teaching of introductory programming and have also suggested seven principles of programming language design which may help to reduce those hindrances. We do not suggest that either list is exhaustive, nor that the principles we expound are definitive or universally applicable. The design of any programming language is an art, and the design of a language whose purpose is to teach the fundamental concepts of programming itself is high art indeed.

Just as no battle plan survives contact with the enemy, no pedagogical language design (no matter how sound its design principles or clever their realization) can hope to survive contact with real students. Yet the outcomes of such encounters are the only meaningful measure of the success of an introductory language. This implies that the most important tool for pedagogical programming language design is usability testing, and that genuinely teachable programming languages must evolve through prototyping rather than springing fully-formed from the mind of the language designer.

References

[1] Wirth, N. and Jensen, K. *Pascal User Manual and Report*, Springer-Verlag, 1975.

[2] Holt, R.C. and Hume, J.N.P., *Introduction to Computer Science using the Turing Programming Language*, Prentice-Hall,1984.

[3] Geurts, L., Meertens, L. and Pemberton, S., *ABC Programmer's Handbook*, Prentice Hall, 1990.

[4] Kernighan, B. and Ritchie, D., *The C Programming Language, 2nd ed.*, Prentice-Hall, 1988.

[5] Stroustrup, B., *The C++ Progarmming Language, 2nd ed.*, Addison Wesley, 1991.

[6] Barnes, J.G.P., *Programming in Ada*, Addison-Wesley, 1994.

[7] Harbison, S., *Modula 3*, Prentice Hall, 1992.

[8] Hudak, P. and Fasel, J.H., *A Gentle Introduction to Haskell*, SIGPLAN Notices, 27(5), May 1992, ACM

[9] Springer, G. and Friedman, D.P., *Scheme and the Art of Programming*, The Massachusetts Institute of Technology, 1989.

[10] Feuer, A. and Gehani, N., *Comparing and Assessing Programming Languages: Ada, C, and Pascal*, Prentice-Hall, 1984.

[11] Conway, D.M., *Criteria and Consideration in the Selection of a First Programming Language*, Technical Report 93/192, Computer Science Department, Monash University.

[12] Koelling, M., Koch, B., Rosenberg, J., *Requirements for a First Year Object-Oriented Teaching Language*, Techical Report 94/488, Computer Science Department, Sydney University.

[13] Mody, R.P., *C in Education and Software Engineering*, SIGCSE Bulletin, 23(3), 1991.

[14] Wilensky, R., *LISPcraft*, Norton, 1984.

[15] Hofstadter, D., *Gödel, Escher, Bach: an Eternal Golden Braid*, Part II, Chapter 10: "Similar Levels", Basic Books, 1979.

[16] Stroustrup, B., *The Design and Evolution of C++*, Addison-Wesley, 1994.

[17] Holt, R.C., Matthews, P.A., Rosselet, J.A., Cordy,J.R., *The Turing Programming Language: Design and Definition*, Prentice Hall, 1988.

[18] Clocksin, W.F. and Mellish, C.S, *Programming in Prolog*, Springer-Verlag, 1981.

[19] Thorndike, E.., *The Fundamentals of Learning*, New York: Teachers College Press, 1932.

[20] Ausubel, D., *The Psychology of Meaningful Verbal Learning*, Grune & Stratton, 1963.

[21] Rumelhart, D. and Norman, D., *Accretion, tuning and restructuring: Three modes of learning*, In. W. Cotton, J.and Klatzky, R. (eds.), "Semantic Factors in Cognition", Erlbaum, 1978.