Notations and Programming Languages

The first thing that one notices about a programming language is its syntax. When we think of COBOL we see lots of capital letters and verbs like "PERFORM". Lisp is associated with lots of parentheses. I have my personal memories of Algol 60 with its wonderful typography with boldface delimiters such as **begin** and **end** way before there was boldface on computers! We often hear about graphical programming languages also. These languages all came with various claims about their applicability. For example, COBOL was designed for business applications and Lisp for artificial intelligence. How is the syntax of the language connected with its power or applicability? Intentional Software has some very specific views on this.

Let's look at equivalent forms in different languages. For example, consider the conditional statements:

```
IF (A .LT. 10) ...
if A < 10 then ...
test a ls 10 ifso ...
if (A < 10) ...
```

It is clear that the syntactic differences are completely superficial, and it is difficult to see why people would have battled so hard for or against them. Indeed it is disappointing to consider that this list represents more than 40 years of programming language evolution from Fortran through C#.

By the way, the 3rd line is written in the historically important but otherwise little known language BCPL from Bell Labs. It was followed by the language B which in turn was replaced by Kernighan and Richie's C. The joke of the time was to guess whether the successor of C would be called "D" or "P"? Check out this overview of the family tree of programming languages.

Now, if we compare the Fortran IV statements:

```
    IF (A – 10) 2, 2, 1
1   ...
    GO TO 3
2    ...
3  CONTINUE
```

with Fortran 77:

```
IF (A .LT. 10)
  ...
ELSE
  ...
```

we see more than just a syntactic difference. The early so called "arithmetic if" statement awkwardly aped the way hardware expressed the comparison: whether the control should be transferred to label 1 when the difference (a-10) was

to the second, assuming we could correctly follow all the details and paths.

The structured form is much shorter, yet it contains the same information. How is that possible? It is because the "arithmetic if" and "goto" statements have more degrees of freedom (in other words, more parameters) than we need; namely, the three label references after the IF and the one label reference in the GOTO. These could have easily represented a loop or maybe up to three different choices, whether on purpose or, perhaps, by mistake. The earlier form is longer because it has to represent the programmer's intention in the larger parameter space. In contrast, the structured form has fewer parameters – its "degrees of freedom" property is more like the problem at hand, to wit, do something if a is less than 10 and otherwise do something else. Therefore the structured form is shorter. It cannot do a loop, nor can it do more than two things, which means that many opportunities for mistakes are eliminated. There are other forms for doing those things explicitly if that is intended.

We could say that the "arithmetic if" and the "goto" statement in FORTRAN IV are less intentional, and that the structured "if" and "else" in FORTRAN 77 are more intentional. We also note that intentionality has to do with how the degrees of freedom in the problem compare with the degrees of freedom in the notation. We know that we cannot express something with fewer degrees of freedom than what is in the problem itself. But they can be equal: that is the intentional optimum. As the degrees of freedom property in the notation becomes greater than in the problem, we can call the notation less intentional and more implementation-like. The difference can be very large in the case of assembly code, or even machine code. In fact the number of bits used to represent the assembly code for our "if" statement might even be sufficient also to represent a recognizable icon of President Lincoln (756 bits)! Clearly we would be using more bits than the problem dictates.

What is wrong with more degrees of freedom? Nothing if that is what the problem needs: for example, if we are trying to create an iconic directory of the Presidents. But if the problem is to choose between two possibilities (as in "if..then..else"), or indeed for any problem, if we are given more degrees of freedom than we need, we incur costs of navigating in the larger space when we seek our solution and we incur costs of bugs. The bug costs are due to the real possibility that we get lost in the large space. For a given problem, the more parameters needed to express the solution, the greater the chances that one of them will be incorrect.

On the other hand, we also need to recognize that even excess degrees of freedom can be useful in some cases. In the early days of computing, use of assembly code was normal practice. If we know very little about the solution and it is expensive to change the language of the solution, then we had better ensure that every possible solution will be accessible within the same language and thereby benefit from the degrees of freedom. But if we know quite a bit about the solution, then we can reduce risks of incurring costs by shifting to new expressions with fewer degrees of freedom where the structure of the solution can more closely approach the structure of the problem.

So just as "arithmetic if"s were replaced with "if..then..else", the process of elimination of excess degrees of freedom has continued. For example, instead of the common loop pattern:

```
startloop:
   if (!a) goto endloop;
   ...
   if (x) goto endloop;
   ...
   goto startloop;
endloop:
```

we can currently write:

```
while (a) {
   ...
```

One problem with the former loop pattern was that the programmer was required to come up with reasonably unique, but still meaningful names for the labels. These are really contradictory requirements. For example “endloop” would have not worked very well in any real system because it would not be unique. Using “L123” would be unique but probably not as meaningful.

The “while” loop and “break” statement cured those contradictions and eliminated unnecessary degrees of freedom. But why should we stop improvements at the current “while”? We can ask next: “What are we doing with the while and its contents?” Chances are that there is some answer like: we are waiting for something, we are searching for something, we are enumerating
over some set. In each case, we could imagine some construct that would do exactly what we wanted if we had the right degrees of freedom. For example we now see “foreach” in languages like Perl, C# and Java, where the freedom (and burden) to separately identify the loop limits while iterating over a collection has been removed. “Foreach” is more intentional than a traditional “for” loop. Although it adds complexity to a language by introducing some new syntax, it more directly gets to the intended behavior of iterating through a collection.

Advocates of object-oriented languages point out that optimal constructs are already within reach – just define a class and its methods the way you want them. But this works well only in cases where a class is an exact representation of an underlying intent. What about a “while” loop, which is not a class? Certainly a class-based enumerator could be built with some effort,
but then the programmer would have to introduce a name for the method that represents the body and depending on the implementation other names would have to be invented as well. So building a class-based “while” loop enumerator out of some more primitive components would still be moving toward the past, raising the old issues again.

Instead, we should move further toward the problem. Once we have introduced the search primitive that the “while” implemented, we should continue asking questions: “What are we doing with this search?” and create a construct for the intention behind the search. To be able to continue creating new constructs, we have to look beyond classes or even aspects.

We could help this process of creating new constructs move along by making notation and semantics independent. We should be able to simply list the degrees of freedom – the parameters – that are required for a construct. The parameters themselves could be any other constructs with no a priori limitations: expressions, statement lists, declarations, or types. And notations will have to accommodate this flexibility.