On Programming Styles

Niklaus Wirth, 24.3.2008

After listing the restrictions, changes, and extensions of Revised Oberon, I realized that I had forgotten an important restriction. I had forgotten it, because I had already become used to it. It is the reason for writing this memo.

In the early days of computing, there was no such thing as a programming style, and therefore one did not talk about style. Today the term “model” seems to have replaced “style”. The early and implicit model was that of a memory and an instruction sequence operating on the memory. The ultimate reduction of this is the *State Machine*. It is characterized by having a state *s* and performing a sequence of state transitions T:

```
{s := T(s)}                    {s := T(s, in); Out(s, in)}
```

If the machine is to make sense, it must accept input and produce output, we need to introduce an input data sequence and an output operator, as is shown to the right.

Most programming languages are built on this model. *Fortran* (1957), and much more so *Algol* (1958), added structure to the program text. The most important element was the subroutine of Fortran and the procedure of Algol, the former notably excluding recursion. But it was not before the appearance of *Lisp* in 1962 that a style was recognized as a distinctive feature. Lisp was exclusively built on the notion of function. Repetition was replaced by recursion.

```
out := F(in)
```

A function has no state. This was what clearly distinguished Lisp from other languages. But its utmost simplicity was also the quick end of what was termed *pure* Lisp. The absence of state, i.e. of variables and assignments to them, reduced Lisp to a theoretician’s fancy. They were soon added, of course in the disguise of “functions”, and “pure Lisp” became Lisp and InterLisp. Fortran and Algol became known as *procedural* languages, Lisp as *functional*.

Although these languages ultimately feature the same constructs and offer more or less the same possibilities, the distinction between styles persisted and survived. What is their distinctive difference? In my view, the main differences lies in the absence of global variables in functional languages, and thereby of side-effects.

It has become an accepted notion, that the functional style has its merits, not so much because of the functional notation, but for avoiding global variables, and with it nesting and side-effects. The belief has also spread, that the best solution lies in a combination of both styles, accepting their positive and omitting their negative aspects.

The most significant step in this move is to eliminate access to objects that are not strictly local. However, this is going slightly too far. A better option is to allow access to strictly local and to strictly global objects. The latter are typically state variables and they retain the notion of state. Apart from them, the recommended style becomes predominantly functional. With the adoption of these rules, also the nesting of procedures (of scopes) essentially becomes irrelevant.

The first system disallowing access to not strictly local objects was the Algol compiler for the Burroughs B-5000 computer in 1964. The reason then, however, was a simplification of the compiler rather than a methodological one. We have introduced the same restriction in the revision of the language Oberon in 2007.

In summary, the new rule of good style is: Avoid nesting of procedures and use global variables sparingly! Of course this is no dogma, and rules allow for justified exceptions.