**[Tradeoffs in Programming Language Design](https://moodle.cis.fiu.edu/v2.1/mod/page/view.php?id=8947" \o "Tradeoffs in Programming Language Design)**

**by Geoffrey Smith**

Why are there so many programming languages?

Undoubtedly there are many answers. For example,

* Some languages are designed for a specific application domain, and others are general purpose.
* A company may wish to develop its own language for marketing reasons.
* A standardized language is less open to innovative extensions.

But here I want to focus on the *tradeoffs* in programming language design.

Briefly, the idea is that many of the desirable characteristics that a programming language might possess turn out to *conflict* with each other.

This means that a programming language designer must unavoidably favor some of these characteristics at the expense of others, based on the particular goals of the programming language.

Here are some desirable characteristics:

* **expressiveness**

Can the language directly represent a wide variety of computational objects?

* **implementability**

Is it easy to implement the language efficiently, in both time and space?

* **predictability**

Is it easy to reason about how programs in the language behave?

* **uniformity**

Are the rules of what is allowed in the language simple and uniform, or are there exceptions? For example, can *any* type of value be returned by a function, or can only "simple" values be returned?

**Example Tradeoffs**

1. **32-bit integers**

Letting int be limited to 32-bit, two's complement integers is good in some ways, and bad in others:

* + + implementability

Such integers are directly supported by typical processors.

* + - expressiveness

Integers above 2147483647 (which is 231-1) cannot be represented.

* + - predictability

Various algebraic laws *fail* with 32-bit integers.

If overflows cause exceptions, then associativity can fail: (a+b)+c may not be equivalent to a+(b+c). If a = 2147483647 and b = 1 and c = -1, then (a+b)+c causes an overflow, but a+(b+c) evaluates successfully.

If overflows are ignored, then a < a+1 can fail: if a = 2147483647, then a+1 = -2147483648! In addition, the property a\*b = 0 only if a or b are 0 fails:-2147483648 \* 2 = 0!

The design alternative of F# bigints solves these problems with expressiveness and predictability, but hurts implementability.

1. **Side effects**

If an expression can cause changes to memory, then it is said to cause *side effects*.

* + + expressiveness

C idioms like stack[++p]=v; are quite handy.

* + - predictability

If expression e causes side effects, then e+e is usually not equivalent to 2\*e.

For example i++ + i++ is not equivalent to 2 \* i++.

Also order of evaluation becomes significant, as in cases like i++ \* i--. (For this reason, Java specifies left-to-right evaluation of expressions.) (In contrast, C leaves the order *unspecified*.)

1. **Concurrency**
   * + expressiveness
   * - predicatability

We would expect x=2; to be equivalent to x=1; x++;.

But this is not true if the code is run in parallel (with interleaving) with x=3;.

x=2; || x=3; results in x getting value 2 or 3.

x=1; x++; || x=3; results in x getting value 2, 3, or 4!

1. **First-class functions**

If functions are just another kind of value, with all the "rights" of any other value, then we say that the language has *first-class functions*.

For example, F# has first-class functions, as illustrated by the function mk\_expon from our First Taste, which takes as input a times function and a one value and returns an exponentiation function.

* + + uniformity

Not allowing functions to return functions would be a non-uniformity.

* + + expressiveness

As illustrated by mk\_expon, first-class functions are quite useful.

* + - implementabililty

As we will see later in the semester, first-class functions cause significant challenges for implementations.

In summary, a basic reason for the existence of many programming languages is that there is no single optimal design.