5 Mistakes in Programming Language Design

In the recent years, programming language design has gone through a renaissance primarily due to two factors: (1) Multicores became standard in consumer PCs. This brings more attention to concurrent programming. (2) Dynamic languages are fast enough to implement internet services and outgrow the demeaning term "scripting language".

With this article, I try to collect the most important mistakes that every serious programming language designer should avoid. I will ignore opinionated questions like dynamic vs static typing. Also, I will omit mistakes that can be fixed without major pain. For example, adding parametric types later on seems to be possible. Sun added Generics to Java eight years after the 1.0 release. As a recent example the Google Go Language Design FAQ states: "Generics may well be added at some point. We don't feel an urgency for them, although we understand some programmers do."

0. Null pointers

A reference to an object may be a null pointer in all popular languages, which leads to runtime errors. C.A.R. Hoare recently took on responsibility for this "invention", though a lot of other designers are to blame as well.

> *I call it my billion-dollar mistake. It was the invention of the null reference in 1965. [...] More recent programming languages like Spec# have introduced declarations for non-null references. This is*

> *the solution, which I rejected in 1965. - C.A.R. Hoare*

Languages like C/C++ just crash with a segmentation fault on dereferencing a null pointer. Java, Python and other languages throw a NullPointerException, but the problem is that this RuntimeException may be thrown at nearly every statement. A static type system can be used to declare references, which are guaranteed not to be null. For example Cyclone is a safe C variant, which introduces a not-null pointer and restricts pointer arithmetic.

Some languages even make it impossible to create a null pointer, though this makes explicit pointer arithmetic impossible. Haskell for example provides the Maybe monad instead, which forces the programmer to consider the "null case".

1. Parser-unfriendly syntax

The grammar of a programming language should be LALR or even better LL(1). Todays programmers want proper tool support for their language, which means multiple IDEs, editors and other tools have to parse a programming language. There won't be a single frontend. Probably multiple compiler implementations are created. This development should be made as easy as possible. The prominent counter example is C++, which is nearly impossible to parse correctly. The syntax overhead is neglible and the programmer enjoys faster compilation times in return.

2. Unclear semantics

Don't say "implementation specific" in a language specification! Use "undefined" as rare as possible! The gold standard is StandardML, with a completely formalized semantic. C is a counter example with many edge cases unspecified. However, due to its widespread use some behaviour is defined by community consensus. For example it is unspecified what happens on an integer overflow. This means a compiler could correctly deduce `x < x+1` to be always true (maybe a whole loop could be optimized away in effect). Unfortunately a lot of C code uses a comparison like that to check, wether an overflow happened, and expects that an overflow behaves like on a x86 processor (e.g. `maxint+1 == minint`).

Clear semantics make verification and error checking easier. Software verification is coming slowly, but surely. I envision a big chance for languages which make verification easy in a decade or two, which is roughly the time a successful language started today will reach the mainstream.

3. Bad Unicode support

This mostly is about strings within the code, but do not forget that not everybody names his functions in english. The programmer should be able to use Unicode, so the source file must declare its encoding.

The conversion and differentiation between a text and a byte stream is more an issue of the standard library than of the language, but of course this influences the syntax as well. Read how Python 3 fixed this.

4. Preprocessor

While preprocessors like cpp and m4 are in widespread use, they are a quick hack instead of a clean solution. They are used to include external files (without the possibility for proper module mechanisms), enable conditional compilation, and macro substitution (enabling subtle bugs, because of scoping, syntax etc). Integrating these features into the programming languages can improve performance and error checking, but does have no downsides.

A particular bad use of a preprocessor is the module system. C suffers from `#include` and C++ creates even more pain, since templates require big parts of a program to be in header files and included into every other file. With a real module system the `extern` keyword would not be needed and linking would be much faster.

More

This list is not comprehensive and i plan to extend it. Please, tell me what i have forgotten!

Chinese translation, reddit discussion, hacker news discussion.

Also read, how a language can be faster than C.

Thanks for Greg Hluska for editing.