The Problem

There is a class of error that is well understood and can be detected by automation.

This class of errors is known as “syntax errors”.

For this to work[1], the input language must contain *syntax* in the form of *syntactic sugar*.

There are, at least, two kinds of syntax errors:

1. simple typos
2. deeply nested constructs with incorrect nestings.

There are, at least, two kinds of programming languages:

a) Those that provide syntactic sugar that allows syntax errors to be detected
b) Those that eschew syntactic sugar and do not provide (much, if any) syntactic sugar.

Languages in class (a) typically come from the Algol (Pascal) lineage where most structured constructs are clearly delimited by end phrases. For example, languages that provide “if ... end if” fall into what I call class (a).

Languages like Common Lisp and C fall into what I call class (b). In C, an “if” statement ends with a single character “}” and constructs like “for” end with the same symbol “}”. In Common Lisp, all constructs end with the character “)”. It is impossible to know what “}” and “)” pertain to, except by scanning backwards to find the beginning of the corresponding construct. This is harder if the construct is deeply nested and if the beginning of the construct is off-screen (or outside of the window).

Shallow solutions to this symptom (b) have included attempts to make pretty printers and to embed them into code editors.

With the advent of better and better code optimizations (including functional alpha and beta substitution), the editor-debugger combination cannot easily display, correctly, the various constructs.

When is Syntax Checking Important?

Syntactic correctness is important when the code is freshly minted (newly written, alpha-level, not tested yet).

In such cases, automation, in the form of syntax checking, can help weed out this class of errors.

In later stages of coding, syntax checking becomes less important.

---

1 For automatic detection of syntax errors.

At later stages, a programmer becomes less concerned with syntactic correctness of the code (since it has already been checked) and more concerned with expression of the algorithms being used.

In even later stages, programmers new to the code (e.g. Msaintenance Engineers) explore the (already-working) code and try to understand it.

Those who favour languages in class (b) tend to be programmers who place more importance on the later stages of coding and maintenance.

Programming Language Design

It was believed that designing programming languages was a hard problem.

This is no longer the case.

It was believed that processing time was valuable. This affected programming language design – all languages were created with a *single* syntax.

Proposed Solution: The Best of Both Worlds

A solution to the apparent dichotomy of syntax checking via syntactic sugar vs. deeper understanding is to provide two (2) syntaxes for every language – a *writing syntax* and a *reading syntax*.

The *writing syntax* would contain syntactic sugar such as “end if”.

The *reading syntax* would contain no syntactic sugar and would conserve screen real-estate to allow for easier understanding of details.

In some cases, syntactic sugar could help maintainers understand the structure of unfamiliar code. Programming editor(s) could assist the human reader (developer, maintainer) by switching between *writing* and *reading* syntaxes on command.

DSLs could be used to restructure programmers’ workflows, e.g. a front-end DSL could perform syntax checking and generate syntactic-sugar-free source code for later understanding and processing.

Divide and conquer means that once some aspect of a problem is understood, that aspect should be chopped off and isolated, leaving a “simpler” problem to be solved. When it comes to language design, we can apply *divide & conquer.*

1. We already know how to detect one class of problems in a language. It is well-understood how to create syntax and parsers for a reasonable set of programming languages, and
2. The rest of language processing remains a “hard” problem. [In fact, the problem of generating good, portable code was explored and is also well-understood. What’s left? Maybe we already know how to make semantic sense of a program (e.g. Denotational Semantics)? Most previous attempts tried to solve the *whole* problem in a *single* notation. Peter Lee[2] and Uwe Pleban made

2 https://www.amazon.ca/Realistic-Compiler-Generation-Peter-Lee/dp/0262121417

interesting progress on these fronts, but tried to define all passes using a single tool instead of using multiple tools and multiple paradigms.]