2024-03-31 Free Range Programming

Buggy Software	2
Parsing with Ohm and PEG	5
Sequentialism Gone Wild	6
Making Everything Visible and Explicit	9
Statement Sequencing	10
Simplicity	13
Appendix - See Also	14

Buggy Software

Q: Why do end-users need to protect apps from one another?

A: because programmers release buggy software and have successfully conditioned end-users to pay for the privilege of using buggy software, and, have conditioned end-users to act as free (actually negative expense) Q/A departments.

Programmers have conditioned end-users to pay for expensive hardware and software to be able to use buggy software.

Software was buggy 50 years ago.

Software is still buggy today, maybe even more buggy, and, the software culture creates more attack vectors for criminality than was prevalent 50 years ago.

What's wrong with this picture?

Hand-waving arguments about "essential complexity" will not be accepted. The point of researching these problems is to understand complex phenomena, then, to explain the phenomena in simple-to-understand terms. Understanding the phenomena, then explaining the phenomena in obtuse forms is not acceptable.

BNF is Better Than Parser Combinators

The belief that parser-combinators are more composable than BNF, is basically wrong.

It is more convenient to write mini-parsers in a DSL designed specifically for the problem - BNF - and to stuff each one of those bits into separate processes.

It is plain easy to compose programs using processes.

We don't do this because it is mind-numbingly inefficient to use processes, today.

Why is this inefficient? Because of the forced use of function-based notation which forces the use of preemption, which needs extra software to run it. In addition, there are the heavy-handed concepts of MMUs and TRAPs to various protection layers.

Let's say that you get fully deprogrammed from the cult of sequentialism. You are given a choice of using parser combinators or process-based parser tidbits. You would choose to use process-based parser tidbits in processes, because you can snap them together like LEGO® blocks, whereas to snap parser combinators together you face many more restrictions, like the requirement to always produce a result even when you don't really want to (that whole "nil" problem, déja vu all over again).

If you are still affected by the cult, you would eschew this solution because you simply cannot "see" how this can be done efficiently.

Bizarrely, the cult itself, provides the answer - closures.

Processes are closures, but closures are cheaper, by a lot.

Basically, there is a huge chasm between the act of protecting apps from one another and simply writing a single program.

You need heavy-handed processes and MMUs to wrap apps and keep them away from each other - but - you don't need such heavy-handed approaches when you are simply writing a program.

Actually, protecting functions from one another by using processes might be a useful tool for developers, but, is wildly inefficient for inclusion in production code. Developers need <u>different</u> tools and machines than those that end-users are willing to pay for. That's another issue in today's programming culture - the idea that code shipped to end-users <u>needs</u> to use the same stuff as what developers use, e.g. bloated operating systems.

Parsing with Ohm and PEG

By the way, PEG - Parsing Expression Grammars - make BNF much easier to use. PEGs are *waaay* better than REGEXs and *waaay* better than CFGs (for certain quickie things).

My favourite PEG, today, is OhmJS (ohmjs.org). OhmJS makes PEGs much easier to use.

Why?

Firstly, because OhmJS cleanly separates parsing from semantics. Most other PEG libraries tangle the two concepts together. Tangled concepts produce confusion, which produces bugs - and, worse - causes stoppage of being “in the zone” (currently called “flow”, or, maybe “focus”). Flow is just as important in programming as it is in sports. When you get into a flow state, you are more productive. Anything, but, anything, will break you out of the flow state, hence, will reduce your productivity. If you need to pepper your grammar with capture-variable names, that interrupts your flow. If you try to read someone else’s grammar and see it peppered with variable names and semantics code, you find it harder - a lot harder - to understand what the other person intended. This is just plain bad.

If you think that creating a new language needs to take more than an afternoon, then you aren’t thinking “quickie”. REGEX used to be a compiler-only technology, but, was promoted to quickie-use by its inclusion in programming languages, e.g. Python, Javascript, PERL, etc. Ohm makes it possible to *imagine* using better-than-REGEX technology for writing quickies. CFGs, like YACC make this idea seem difficult and cumbersome, hence, when you think that you must use CFGs, the idea of using true-blue parsing DSLs for quickie, knock-off bits of everyday coding, gets summarily discarded.

Sequentialism Gone Wild

Here is a shining example of just how deeply ingrained sequentialism is in the current programming culture.

We might write:

```
x = 5;
y = 6;
```

We know that the two statements are independent and can be executed in any order, as long as x has 5 in it and y has 6 in it the next time they are used.

Now, let's change the code to

```
x = 5;
x = 6;
```

We know that `x = 5;` is essentially useless and can be optimized away.

Why do we know that?

Compilers can perform this optimization only because the language guarantees that `x = 6;` happens after `x = 5;` every time. We must rely on compilers to do this extra work because the language does not allow programmers to say - explicitly - what must happen in what order.

Actually, programmers can change the order of lines, but, there is a global cost to using this kind of tricky, implied notation. Every line has sequencing built into it under the hood.

Making this kind of low-level sequentialism implicit, implies that higher layers of programming, also, need to follow this implicit rule while honouring sequentialism, hence, it is harder to implement other kinds of code than is strictly necessary. You have to break the assumption of implicit sequentialism to be able to deal with the problem in some other way, using some other paradigm. Remove global sequentialism and many other problems become much easier to solve.

Going at it this way results in slow, ad-hoc progress. How many decades went by before compilers were built to capitalize on this feature? To my recollection, it took at least one decade, if not more for `gcc` to appear. When I got my first real job in 1981, programmers programmed in assembler and scoffed at crazies like me who pointed at High Level Languages, like C, as being the future of programming. `Gcc` finally shut them up, but, that was almost a decade later. It took even more time for other compilers to follow suit.

Let me emphasize: I'm not advocating for replacing *one* set of rules (sequentialism) with another *single* set of rules (asynchronosity). I'm saying to use *both* sets of rules and to let the Software Architect decide which set of rules best expresses the paradigm-du-jour for solving each aspect of a larger problem space. A problem is solved by nipping away at it using multiple paradigms.

Forcing the use of one and only one paradigm - sequentialism - is not the same as multi-paradigm programming. If a programmer wants to apply a different paradigm to solving some aspect of the problem, the programmer, first, has to waste time creating work-arounds to break out of the ingrained paradigm.

Forcing the use one of only one paradigm results in a game of whack-the-mole. The chosen paradigm solves one aspect of the problem beautifully, but results in fugly work-arounds for solving other aspects.

One fugly work-around that jumps right out at me is the epicycle called “preemption”, which brings with it baggage, like the hand-wringing about "thread safety". Functional programming is the way to use CPUs for expressing compute-ations, but is *not* the way to use CPUs for other kinds of problems.

There *are* other kinds of problems, and, those kinds of problems are becoming more and more important.

It used to be the case that we could only imagine using CPUs for creating ballistics calculators for the military, but, today, we need to deal with internet, robotics, etc., all of which are *not* inherently compute-ations.

We keep proving - slowly and grindingly - that you *can* convert those kinds of problems into compute-ations, but, we *ignore* the idea of *proving* that using the

compute-ation paradigm is the best choice for those cases. Just because you *can* do something one way, doesn't mean that you *should* do it that way.

The idea - and attendant tribulations, gotchas and baggage - of multi-core CPUs is but a manifestation of sequentialism gone wild.

A good Software Architect's toolbelt contains many paradigms.

Starting *every* project with only *one* basic paradigm - sequentialism - ties the Architect's hands and causes unnecessary work.

Paradigms should be free of one another, but, with the current crop of languages, all paradigms are forced to boil down to a substrate layer of underlying sequentialism.

CPUs are - by design - sequential, but, solutions are not *forced* to use CPUs every time. For example, I look at the LED display of my laptop. I am hard-pressed to imagine why I would bother to use a CPU in designing such a display. Yet, the display is an integral part of the thing I call a "computer" [a bad name for that thing, by the way, since I do a lot more with it than just compute-ing, e.g. I move the mouse, which isn't inherently a compute-ation].

Making Everything Visible and Explicit

Denotational Semantics is the idea of using Functional Programming to describe the details of compilation.

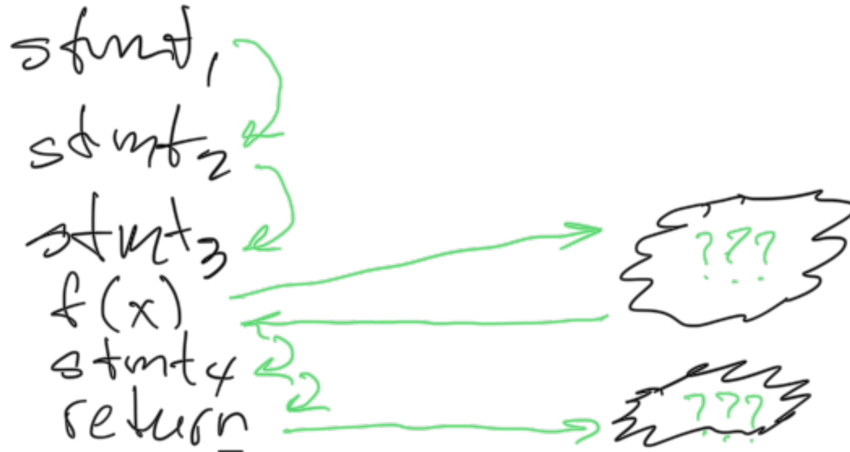
It is important to note that “control flow” is handled in DS by supplying an explicit parameter - the “environment” - to *every* function call in the DS specification.

It is this extreme explicitness that makes DS work.

It is the lack of this kind of extreme explicitness that causes subtle forms of confusion in popular programming languages and projects.

Statement Sequencing

At the language level, we see control-flow described by implicit sequencing of statements - one after the other, with less frequent breaks in control flow described by function calls and function returns.



In this diagram, the green arrows represent implied control flow. Note that when function calls happen and returns happen, we can't tell - simply by looking at the text, what is meant to happen. Where does control flow go? How long will it take? Are there other function calls hidden in those clouds? How many layers of function calls are there in the parts that we can't see?

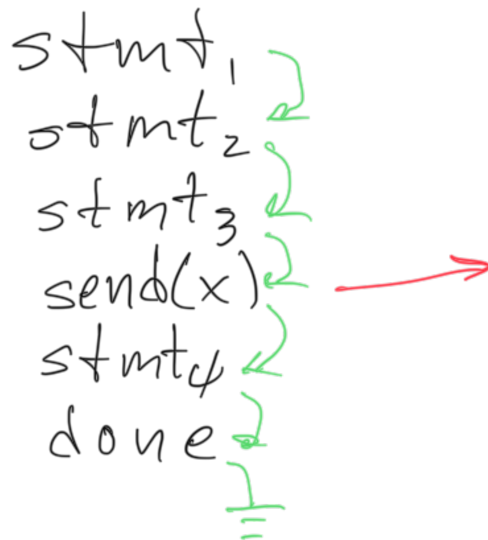
This is an issue of locality-of-reference. You can't tell what's going to happen by looking only at the visible part of the code, so you have to make up some rules that you can rely on. For example, you must have a rule that says "no side effects allowed" so that you can know that calling f(x) did not produce unexpected control flows and did not produce unexpected data in unexpected locations.

But, side effects are what CPUs do for a living.

How can we deal with that reality?

One way is what we currently do - we simply decree that nothing bad can happen when f(x) is called.

Another way, which is mostly ignored, is to turn the outgoing arrow from f(x) into some kind of other arrow - a fire-and-forget arrow. Delete the wait-for-return arrow and turn it into a sequencing-to-the-next-instruction arrow.



Now, we can see *all* of the code involved. The decree becomes simpler - nothing can interrupt the flow of the green arrows. Even when a red arrow shoots something into the ether, we know that we don't need to wait for a result *and* we know that nothing can break our flow through this code, even if the red arrow causes something to fire back at us. If we do get something fired back at us, it doesn't interrupt us, whatever it is just gets queued up for processing at some unspecified time in the future.

'Done' doesn't need to "return" anything, it just quits. If we want to "return" results, we simply fire a red arrow. In fact, we can fire more than one red arrow, or, we can fire absolutely no red arrows. That gets rid of the nil/maybe-sometimes-nil issue.

Functions don't work this way. Function-based programming, including the current manifestation called FP (Functional Programming) doesn't work this way.

State machines work this way. Actors work this way (unless implemented as functions using the sequentialism paradigm). Statecharts work this way. 0D works this way. In fact, the 0D paradigm can be thought of as communicating state machines.

Broadcasting red arrows, willy-nilly to anyone interested in them becomes an unscalable nightmare. The fix is easy. We just need to do what was done with GOTOs. Invent Structured Message Passing. Guess what - somewhere in my

blogs, I show an idea for Structured Message Passing. Somewhat ironically, we see Structured Message Passing all around us every day. In successful businesses, this concept is called “ORG Charts”.

Pub/sub is the idea of broadcasting, willy-nilly, to anyone interested in red arrows. I expect the future of pub/sub to be lathered with more layers of work-arounds, or, better, to have pub/sub simply wither away into obscurity.

Simplicity

Q: Are today's programming techniques causing simplicity?

Or, are today's programming techniques causing complexity?

Have we progressed so much since 1950 in the field of programming that we can, finally, provide guarantees on our software products?

Or, do we need to hide behind fine-print EULAs?

Real Engineering professions, like bridge design Engineers, are regulated by Law. Real Engineers must put guarantees on their work (detailed designs) by affixing their signatures / stamps on their products. If they break their guarantees, they get sued, or, they go to jail.

Is programming there yet?

When Real Engineers face lawsuits and imprisonment, they think differently. They design-in safety factors and big margins for error.

Real Engineering work products cannot be crashed by simple retraction of a small part from "the internet repositories".

Real Engineering work products don't suffer huge spooky-action-at-a-distance bugs caused by unexpected interactions when they, themselves, make slight modifications to their designs.

Appendix - See Also

See Also

References *https://guitarvydas.github.io/2024/01/06/References.html*
Blog *https://guitarvydas.github.io/*
Blog *https://publish.obsidian.md/programmingsimplicity*
Videos *https://www.youtube.com/@programmingsimplicity2980*
[see playlist "programming simplicity"]
Discord *https://discord.gg/Jjx62ypR (Everyone welcome to join)*
X *(Twitter) @paul_tarvydas*
More writing *(WIP): https://leanpub.com/u/paul-tarvydas*