2024-04-18-Free Range Programming

2024-04-18-Free Range Programming 1
Parsing Diagrams 2
What is Programming? 2
Hoary Details Avoided 2
Transpile Pipeline 3
Options Kill 3
Psychology of Notation Matters 5
Ignoring the 4th Dimension is a Bad Idea 5
Appendix - See Also 8

Parsing Diagrams

I use click-bait language on purpose. In an effort to underline important, but, glossed-over ideas.

My current hobby horse is to point out that syntax is cheap, but, paradigms matter.

Parsing diagrams to code is easy even using current technology. Zac Nowicki knocked off a diagram parser in just a few days. I guess that I pointed out what the important syntactical bits are (named boxes with ports, arrow between ports, connections are triples { [down/across/up/through], {from-ref,from-port}, {to-ref,to-port} }, then, Zac provided an XML parser of draw.io's graphML and he provided an interpreter of the resulting data structures. [All in a non-garbage-collected language, yet (his code has since been ported to a GC'ed language (Python))].

What is Programming?

Knowing that you have the freedom to choose syntax "frees your mind" to think about heavier concepts. "Programming" is about making a machine do something useful. "Programming" need not be restricted to a text-only representation. Just knowing *that* tiny detail frees you up to think up newer ways to command machines. Using only text and lambdas and synchrony for building programming languages puts you into a box where everything looks like a compute-ation, making some things much harder to think about than is necessary.

Hoary Details Avoided

CEOs scribble on whiteboards and napkins, then tell their engineers to "make it so".

Most people, including CEOs, don't want to know the details behind REMs[1].

Only a select few care to go into such detail - us.

---

[1] Reprogrammable Electronic Machines. Usually called "computers". I believe that the term "computer" indicates too strong a bias towards compute-ing in lieu of other kinds of uses that Reprogrammable Electronic Machines can be put to.

There are other smart people, people who have been certified smart, like doctors, who care more about details other than the innards of REMs. They need accessibility to REMs that doesn't break their *flow* when they want to think about what *they* think is important.

Silly questions, like "which file / directory do you want to save this in?" Are flow-breaking interruptions.

When you write ideas down on *paper*, the medium - paper, notebooks - doesn't ask you silly, detailed, out-of-band questions, it just lets you follow your line of thought without interruption.

The epitome of useful computing is the "spreadsheet". The idea for spreadsheets started out as a ledger for accountants, but, blossomed into other unanticipated uses, such as personal databases, etc. Why? I would argue that the main reason is that a spreadsheet is a dumb, yet simple-to-use medium that makes sense even to grade-school kids who don't have PhDs.

What other kinds of computerized media are waiting in the wings? What are the low-hanging fruits?

Blender[2] is a "pad of paper" for thinking in 3D. It can create and crank through equations without the user needing to specify concepts as equations.

Flash is a "pad of paper" for thinking in 4D (x,y,z,t). Z is faked-out in a crude way - sprites. Time (t), is made explicit through the use of a timeline editor - crude, yet, usable.

Descript ties Whisper A.I. transcriptions-with-timestamps into a video-editing tool based on word processing instead of clunky timeline editing.

Transpile Pipeline

The first thing I did with OhmJS when I finished the bulk of my learning curve was to create a DSL for writing DSLs in OhmJS, using (of course) OhmJS.

Options Kill

[2] the 3D graphics tool

A bag containing every possible option within a program, is just the half-way point towards making a useful piece of software. Users don't want options, they want all of that power, but, they don't want to have to deal with the niggly details.

My current ideal tool set is something like Lisp plus (+) OhmJS. Lisp is an ever-growing toolbox of paradigms written in a normalized, regular syntax[3] that only machines can love. OhmJS, on the other hand, is a way to knock off little languages that present useful views on slices over the larger set of options. E.g. OOP is one "syntax" that restricts you into thinking in terms of objects with capabilities, without needing to worry about the details of how those objects are constructed. Prolog is another "syntax" that lets you express "exhaustive search" in a convenient manner, but, makes expressing other simple ideas difficult - like simply formatting a string (Javascript is better at that aspect). Statecharts, and Drakon, are a "syntax" aimed at describing control-flow. DPLs (Diagrammatic Programming Languages) are a way to express concurrency.

[3] Lisp is really just an assembler with recursive, instead of line-oriented, syntax. Programmers write ASTs (actually CSTs, but, I quibble) and skip any pretence of using a "syntax".

Psychology of Notation Matters

Ignoring the 4th Dimension is a Bad Idea

Explicitness is good, but cumbersome.

Implicitness is OK, but often turns into a Religion when people forget what the basic simplifying assumptions were - the assumptions that allowed some details to be elided away and to be made implicit.

The Big Win in Denotational Semantics[4] is the idea of making *everything* explicit. At the time that Denotational Semantics was invented, the concept of "environment" was considered to be a "given" and was buried in implementation details[5]. In essence, "environments" were "global variables". Bad. Handling environments explicitly as parameters, though, caused the concept to be normalized and let us (the collective "us") see the forest instead of the individual trees.

It seems obvious to me that, today, we are sloughing off "time" as being an implicit global variable. We know that it's there, but, we keep avoiding it and we keep making it implicit and, therefore, we keep handling the concept in an ad-hoc manner. "Time" is ubiquitous. We see "time" everywhere. Even in Bret Victor's "Stop Drawing Dead Fish" demo (the fish wiggles it tail, the fish squishes up before shooting off to stage-right, etc.).

How was Victor's demo done? I would guess that it used ad-hoc, imperative code with the concept of *time* sprayed throughout the code internally.

Statecharts are a step towards the formalization of time-based sequencing.

---

[4] Peter Lee's book "Realistic Compiler Generation" Is my favourite down-to-earth exposition of the concepts of Denotational Semantics. See https://guitarvydas.github.io/2024/01/06/References.html.

[5] In fact, John R. Allen's "Anatomy of Lisp" is the first place I saw explicit handling of "environment". I built a whole Lisp compiler based on the code in that book (with judicious bug-fixings) in just 22K of memory on a Z80 8-bit REM. I had enough room left over to be able to compile *iota()* [I was an APL programmer by day]. The compiled version of *iota()* ran *much* faster than the interpreted version. A complete language interpreter plus a language compiler plus a garbage collector fit in my 8-bit machine, in only 22K (Kb, not Mb, not Tb) of memory.

Pipelines are a step towards the formalization of time-based sequencing.

Functions, function CALL and RETURN, are a step backwards. Functions elide the notion of time so that you don't even see it. To think about *time* you have to treat the concept as a second-class citizen and write it out explicitly. Most programmers don't bother to do this and the concept becomes lost and smeared throughout their code in an ad-hoc manner.

In fact, functional notation denies the existence of *time*. The simple function call syntax `f(x,y,z) -> q,r` says that there is only *one* input parameter which is deconstructed into three parts and that there is only *one* output result which is to be deconstructed into two parts. All of the inputs - x,y,z - are provided at the *same* time, i.e. time doesn't matter, since we can replace functions faster than the speed of light without regard to the Physics of Reality (that happens to be called "referential transparency"). This simplified notation works OK for problems that are compute-ations, but, this notation is clumsy for problems that are sequenced in time, like servers, clients, robots, GUIs, blockchain, etc.

Maybe we need syntax more like:

```
      f(x)(y)(z) -> (q)(r)
```

Where separate parameter lists and return lists are used to remind us that parameters can arrive in any order.

The above syntax does not remind us, though, that a certain parameter might arrive more than once, say 2 x's before any y's or z's.

One classical interpretation of data flow is that a function fires only when all parameters have arrived. Such an interpretation implies a certain behaviour that constrains the program to a single paradigm. This classical paradigm, unnecessarily, constrains how f can behave - it must wait until every parameter has arrived.

At the most basic level, we want to fire the function - f - every time any parameter arrives, thus, letting the function decide whether to enforce a certain parameter order by declaring errors if necessary. At this basic level, it becomes f's responsibility to perform input validation, instead of allowing the notation to build in such enforcement.

Additionally we want to leave the decision of *when* to produce output to f and *where* to send that output. Traditionally, functional thinking restricts the behaviour of functions such that they are always required to produce an output given an input (resulting in the need for *nil* values and the like), and, functions restrict routing decisions, i.e. a function must always return a value to the caller. A derisive name has been coined for functions that make their own routing decisions - *side effect*. Avoidance of side effects is useful when describing compute-ations, but, does not apply to units of software such as *daemons.* To use function-based thinking to express the operation of *daemons,* programmers must stop and invent work-arounds (aka “epicycles”).

Inputs come whenever they come, and no blocking is implicitly required, i.e. asynchronous, not just step-wise simultaneous.

On the other hand, why bother with this kind of textual notation? DPLs express these concepts more conveniently, as we can see in whiteboard and napkin-based diagrams. Textual notation is a convenient way to express compute-ations, and should be used for only that kind of expression, without being stretched beyond the sweet spot of textual notation’s abilities. Textual notation can conveniently express concepts that are cumbersome to express in DPL form, but, that doesn’t mean that textual notation should be the *only* notation used.

Formalization is hard. Don’t let formalization drive practice. Recognize when formalization is being stretched beyond its bounds. We need more research on how to recognize when a notation is being over-stretched. Early warning signs - “tells”. For example, maybe “difficulty”

1. concurrency is considered difficult to deal with

2. operating systems were invented to insert re-entrancy into REMs where re-entrancy did not exist

3. 50+ years of research is being measured incorrectly. In the 1970’s one could build complicated software (e.g. games) that fit in Kb’s of memory. In 2024, software is no more robust, yet, needs Mb’s of memory, i.e. software has become bloated.

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*