2024-04-03 Free Range Programming

Goal

The main question that I'm interested is is: Why is programming software so hard? What can be done to simplify it - drastically (10x or more simpler)?

I believe that we are working in a realm of FoPoC - Future of the Past of Coding.

To grease the creative juices and to attack FoP - Future of Programming - with greater speed and fluidity, we need more convenient notations for programmers that don't brain-lock them into a single paradigm.

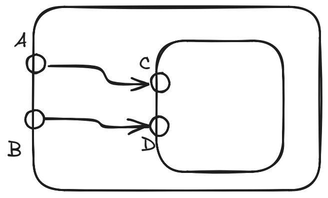
Current PoC - Past of Coding - dictates that everything must start out using the synchronous paradigm (aka "sequentialism"). This paradigm is good for building calculators, like complex ballistics calculators for the military. But, this paradigm discourages thinking along the lines of time-based operations, like DAWs, iMovie, Visual Programming, Actors, internet, concurrency, robotics, etc., etc. It is *possible* to force the synchronous paradigm onto these other kinds of problems, but, it is *inconvenient* and, thus, discourages rapid advancement in these directions. The fact that we need to use *preemption* is a *tell* that we're forcing an ill-fitting notation onto problems.

Normalization

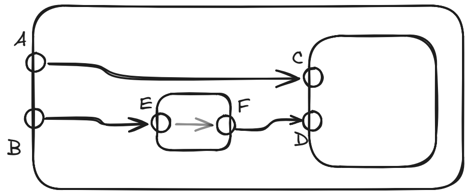
- normalize to simplify
- Eschew complications, like data structures, magic characters, over-specification of details (until we need to deal with such details), etc.
- Example: UNIX® File Descriptors - everything, including devices, look like “files” and can be manipulated through the same, simple interface: open(), close(), read(), write(), etc.
  - UNIX revolutionized programming, especially Operating Systems
  - UNIX® provided a relatively small, but highly reusable tools for text processing
  - The lessons of UNIX® have essentially been forgotten for statement-level programming, due to UNIX®’s association with operating systems
  - UNIX® processes are merely clumsy implementations of closures (see Greenspun’s 10th Rule). All modern languages implement closures, so we don’t really need bloated operating systems any more.
- Example: UNIX® pipes
  - Pipes for text rely on a magic character (newline) -- an unnecessary complication which constrains thinking. Most UNIX® commands are meant to manipulate text, but, at lower levels, UNIX® processes can manipulate other kinds of non-textual data. Then, there’s the question of “why bother?”, maybe t2t (text to text) is “good enough” for most of our problems. Function-based programming (of which, FP, is only a subset) conflates the issues and makes it harder to imagine more innovative uses for programs. If you choose to use a CPU, you *must* deal with it on its own turf - i.e. sequentialism. This does not mean, though, that your higher level notation needs to be sequential.
  - /bin/sh, /bin/bash, /bin/zsh, etc. understate the power of the “pipe” normalization, due to insufficient syntactic power (i.e. textual, stdin, stdout, stderr

- Needs a Visual syntax - VSH (Visual SHell)

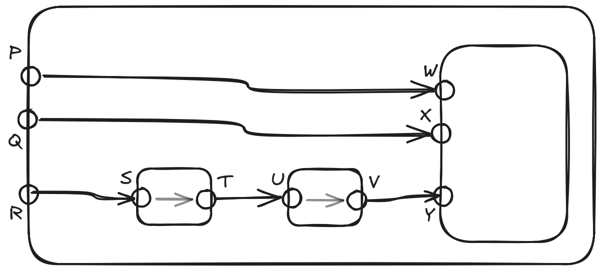
Slip - TDP Timing Dependent Programming



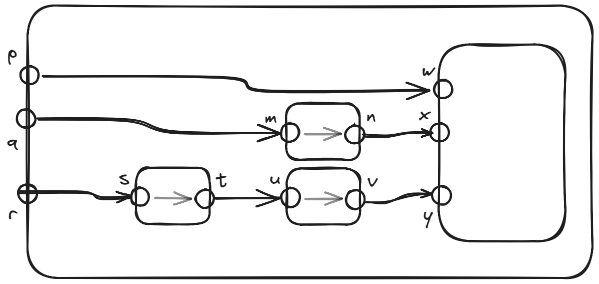
C & D Fire
but we don't know
in which order



C & D Fire
it is guaranteed that
C fires before D



Y is guaranteed
to fire only after,
both, W and X fire,
but, the order in which
W and X fire is indeterminate



x is guaranteed
to fire after w,
y is guaranteed to
fire after x

What are the little boxes called?
Hesitate, lull, rest, stutter, slip, clutch,
decoupler, slip ring, skid

Syntactic Composition

- normalize all communications between components to be “little languages” (as strings)
- Use OhmJS to inhale and parse incoming strings.
- Convert the parsed info into internal data structures and internal representations that are useful for solving one slice of a problem.
- Solve the sub-problem, exhale another “little language” as a string to the downstream components in the pipeline.
  - For example, a compiler:
    - Pass 1: Scanner - inhale human-readable text and parse it with OhmJS (syntax #1), tokenize it and exhale a machine-readable little language containing tokens. The exhaled little language does not need to look “nice” to humans, it just needs to look “normalized” to the OhmJS parser in pass 2.
    - Pass 2: Parser - inhale little language containing tokens and parse using OhmJS (syntax #2) and check that the tokens constitute valid syntactic sequences, i.e. check for “syntax errors”
    - Pass 3: Semantic analysis 1 - use OhmJS to parse incoming text (syntax #3), then, gather up declarations and create an internal symbol table. Exhale a different little language that is annotated with symbol table information.
    - Pass 4: Semantic analysis 2 - use OhmJS to parse incoming text (syntax #4)

Rules of Message Passing (0D)

- Containers
- Leaves
- SMP - Structured Message Passing
  - See “Org Charts”
- Down, up, across, through

RWR - Text Rewriting

- simpler form of Term Rewriting

DI - Design Intent

The goal is to communicate design - quickly and easily.

The goal is to allow push back on the design and, if someone likes the design ideas, to steal the ideas and to reuse them elsewhere.

A notation is measured by how easily it communicates ideas between humans.

If you want to communicate scripts to CPUs, you *must* use assembler. Period. By definition.

Haskell, for example, ultimately compiles down to assembler. The CPU doesn’t care that you used Haskell. You, and other developers, are the only ones who care that you used Haskell.

Can we come up with alternate notations from Haskell for communicating between humans?

Can we use more than one notation for communicating between humans? For example, using Haskell for expressing complex calculations (military ballistics, crypto) plus using StateCharts for expressing scripting recipes plus PEG for expressing text parsers. Maybe more notations? A 2-type language for playing around with ideas and refining them incrementally vs. a zillion-type, static language for expressing optimization and Production Engineering.

It seems that the first choice of just about everyone is to use sketches drawn on napkins at diners, or, whiteboards in meeting rooms.

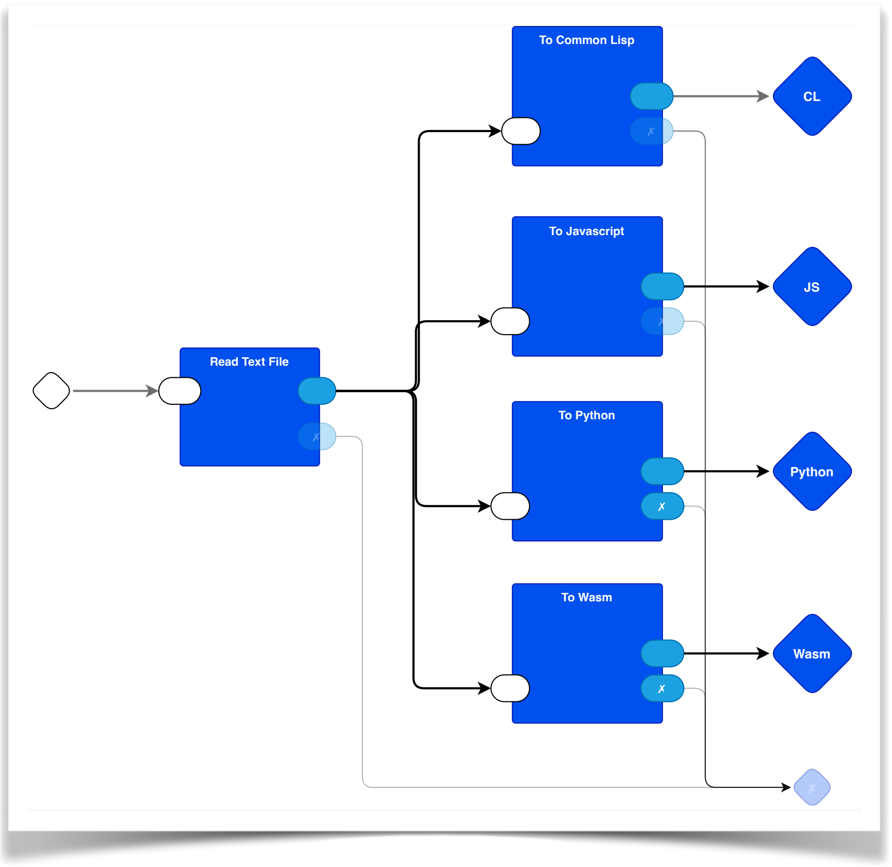
Can we make it such that those sketches contain enough “high level detail” to be refined into scripts for running machines?

Can we preserve provenance from the original airy-fairy sketches down to the actual scripts that are fed into machines?

Code is cheap, thinking is hard.

IR - Intermediate Representations

With respect to the `arith0D` demo repository (Python POC: https://github.com/guitarvydas/0D/tree/devpy/python/test and Odin: https://github.com/guitarvydas/arith0d)...



The ultimate goal is to use Python/JS/CL/WASM as *assemblers* and never have to bother writing in such low-level 3GLs again.

The IR should be an HHL (higher-than-high-level language) that should allow easy machine manipulation and should allow easy transformation into other syntaxes.

Syntaxes are cheap, paradigms are hard.

Clojure is wildly more complicated than Lisp-without-macros. For example, in Lisp, you write recursive syntax using parentheses, in Clojure you need parentheses and square brackets (that's 100% more complicated :-).

The IDEs for Common Lisp are much more mature than the IDEs for any modern language, including Clojure. A significant portion of the goodness of Lisp has been lost due to the push for static compilation and type checking. The “dynamic” parts of Lisp, and the REPL, are good for iterative design refinement, whereas static compilation is good for Waterfall development of super-optimized code for Production Engineering activity. A REPL for a statically-typed-and-compiled compiled language isn’t even close to being as useful as a REPL for a dynamically-typed language (or, even an untyped language, if you wish).

Ideally, the IR should allow you to work with any language and IDE, so it should just be a matter of choice and familiarity.

I haven't settled on anything yet, but, I admire Sector Lisp and OhmJS for their simplicity.

OhmJS makes it possible to use just about any syntax, not just that of Lisp.

The advantage of Lisp's "syntax" is that it allows recursive expression of syntax. I.E. Recursion at the syntactic level, not just at the runtime level. And, it is easy to parse using ad-hoc techniques (recursive descent) or tools like OhmJS and PEG.

Recursive syntax is especially easy to parse with OhmJS (and other PEGs) because PEG can parse matching brackets. CFGs, like YACC and stuff in the Dragon Book, make it cumbersome to parse that kind of stuff. REGEX is even worse - it lulls you into thinking that you can parse stuff with it, but, ultimately wastes your time when you try to scale up to more interesting grammars.

Additionally, in PEG, you can say "and skip this stuff as long as the brackets match", whereas in CFGs, simple ideas like that turn into mega-projects which tend to be avoided.

I think that Lisp has zero (0) syntax. Lisp is assembler with a recursive syntax, whereas Assembler is assembler with a line-oriented syntax (flat, non-recursive syntax).

In Lisp, you are forced to write programs by writing ASTs. Some people love this freedom, others hate it.

RT

I began thinking about an über-syntax - an “IR” if you wish - that I was tentatively calling “RT” (Recursive Text).

I haven’t settled on any yet, but, I currently think that it will take the form:

《*operator operand operand...*》

Where the brackets are hard-chosen to be Unicode, to leave valuable ASCII characters alone. Unicode has so many characters that we can afford to choose some and assign fixed meanings to them, without using up valuable character-space real-estate.

Note that Ohm (PEG) allows parsing arbitrary strings of characters (as long as there is something unique in the strings, to differentiate them), so we might write stuff like:

《I want a *hamburger with 2 patties* and *ketchup* and *mustard*》

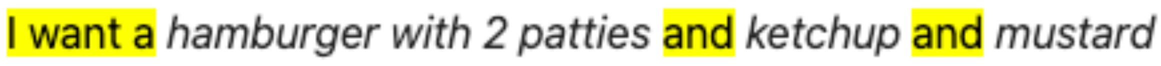
Where italics are use to represent *operands*. The above phrase might be transpiled into:

kitchen.order (burger, 2 * patty, [ketchup, mustard])

Which might be further reduced to something like:

(order kitchen (‘burger (* 2 ‘patty (list ‘ketchup ‘mustard))))

The fact that OhmJS can be used to parse XML / HMTL / etc. might gives us syntax like:



I want a *hamburger with 2 patties* and *ketchup* and *mustard*

which looks - in machine-readable text form - as:

```
<mark>I want a</mark> <i>hamburger with 2 patties</i> <mark>and</mark>
<I>ketchup</i> <mark>and</mark> <I>mustard</i>
```

And, of course, we could parse even more interesting syntaxes composed only of HTML (and CSS) or project-specific syntaxes based on XML / HTML / etc.

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*